DYNAMIC INTERVAL DETERMINATION
FOR
PAGELEVEL INCREMENTAL CHECKPOINTING

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology

in

Computer Science and Engineering

By

Satish Reddy A.

Department of Computer Science and Engineering

National Institute of Technology

Rourkela

2007

DYNAMIC INTERVAL DETERMINATION FOR PAGELEVEL INCREMENTAL CHECKPOINTING

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology

in

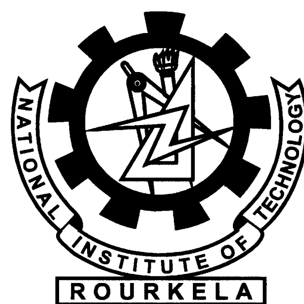
Computer Science and Engineering

By

Satish Reddy A.

Under the Guidance of

Dr. D. P. Mohapatra



Department of Computer Science and Engineering

National Institute of Technology

Rourkela

2007

National Institute of Technology
Rourkela

CERTIFICATE

This is to certify that the thesis entitled "**Dynamic Interval Determination for Pagelevel Incremental Checkpointing**" submitted by Sri **Satish Reddy A** in partial fulfillment of the requirements for the award of Master of Technology Degree in Computer Science and Engineering at the National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University/ Institute for the award of any degree or diploma.

Date:

Dr. D. P. Mohapatra
Assistant Professor
Dept. of Computer Science and Engg.
National Institute of Technology
Rourkela -769008.

ACKNOWLEDGEMENTS

No thesis is created entirely by an individual, many people have helped to create this thesis and each of their contribution has been valuable. I express my sincere gratitude to my thesis supervisor, Dr. D. P. Mohapatra, Assistant Professor, CSE Dept., for his kind and valuable guidance for the completion of the thesis work. His consistent support and intellectual guidance made me energize and innovate new ideas. I am grateful to Dr. S.K. Jena, Professor and Head, CSE Dept for his excellent support during my work. I am also thankful to Prof B.Majhi, Dr. S.K.Rath, Dr. A.K.Turuk, Dr. R.Baliarsingh, Mr. B.D.Sahoo of CSE Dept., for providing me support and advice in preparing my thesis work.

Thanks to my classmates shetty, satish, tony, media, bind, puthal, kalpana, sunil, jayadev, praveen and Gularam for their support and round the clock entertainment for these two years.

Last, but not least I would like to thank my parents for supporting me to do complete my masters degree in all ways.

Satish Reddy

Contents

1 INTRODUCTION **1**

1.1 DISTRIBUTED SYSTEM 2

1.2 FAULT TOLERANCE IN DISTRIBUTED SYSTEM 3

1.3 MOTIVATION 6

1.4 OBJECTIVES OF THESIS 7

1.5 ORGANIZATION OF THESIS 7

2 BACKGROUND AND RELATED WORK **8**

2.1 ASPECTS OF CHECKPOINTING 9

2.1.1 Frequency of Checkpointing 9

2.1.2 Contents of a Checkpoint 10

2.1.3 Methods of Checkpointing 10

2.2 OVERHEADS OF CHECKPOINTING 10

2.3 CONSISTENT SYSTEM STATES 11

2.4 ISSUES IN CHECKPOINTING 12

2.4.1 Orphan Messages and Domino Effect 12

2.4.2 Lost Messages 13

2.4.3 Problem of Livelocks 14

2.5 DIFFERENT TYPES OF CHECKPOINTING PROTOCOLS 15

2.5.1 Independent Checkpointing Protocols 15

2.5.2 Coordinated Checkpointing Protocols 16

2.5.3 Communication Induced Checkpointing Protocols (CIC) 19

2.5.4 Log based Recovery Protocol 22

2.6 REDUCTION OF CHECKPOINTING OVERHEAD 25

3 A CHECKPOINTING ALGORITHM FOR REDUCING COORDINATION OVERHEAD 28

3.1 PROPOSED ALGORITHM 29

3.2 PERFORMANCE EVALUATION 32

4 DYNAMIC INTERVAL DETERMINATION FOR PIC 34

4.1 BASICS OF PROBABILITY 36

4.2 NOTATIONS AND ASSUMPTIONS 37

4.3 EXPECTED EXECUTION TIME WITHOUT CHECKPOINTING . . . 38

4.4 EXPECTED EXECUTION TIME WITH PAGE LEVEL INCREMENTAL CHECKPOINTING 40

4.5 DYNAMIC INTERVAL DETERMINATION 41

4.5.1 Cost Analysis of Expected Recovery Time 41

4.5.2 Interval Determination Mechanism 43

4.6 PERFORMANCE EVALUATION 44

5 CONCLUSION AND FUTURE WORK 48

5.1 THESIS SUMMARY 49

5.1.1 A checkpointing algorithm for reducing coordination overhead 49

5.1.2 Dynamic interval determination for pagelevel incremental checkpointing 49

5.2 FUTURE WORK 50

5.2.1 Coordinated checkpoint algorithm 50

5.2.2 Dynamic interval determination for pagelevel incremental checkpointing 50

Bibliography 51

Abstract

A distributed system is composed of multiple independent machines that communicate using messages. Faults in a large distributed system are common events. Without fault tolerance mechanisms, an application running on a system has to be restarted from scratch if a fault happens in the middle of its execution, resulting in loss of useful computation. Checkpoint and Recovery mechanisms are used in distributed systems to provide fault tolerance for such applications. A checkpoint of a process is the information about the state of a process at some instant of time. A checkpoint of a distributed application is a set of checkpoints, one from each of its processes, satisfying certain constraints. If a fault occurs, the application is started from an earlier checkpoint instead of being restarted from scratch to save some of the computation. Several checkpoint and recovery protocols have been proposed in the literature.

The performance of a checkpoint and recovery protocol depends upon the amount of computation it can save against the amount of overhead it incurs. Checkpointing protocols should not add much overhead to the system. Checkpoiniting overhead is mainly due to the coordination among processes and their context saving in stable storage. In coordination checkpointing, for taking single checkpoint, it will coordinate with other processes. Checkpoint initiating process coordinates with other processes through messages. If more number of messages are used for coordination then it increases the network traffic. Which is not desirable. It is better to reduce the number of messages that are needed for checkpoint coordination. In this thesis, we present an algorithm which reduces the number of messages per process, that are needed for checkpoint Coordination and there by decreasing the network traffic.

The total running time of an application is depend on the execution time of the application and the amount of checkpointing overhead that incurs with the application. We should minimize this checkpointing overhead. Checkpointing overhead is the combination of context saving overhead and coordination overhead. Storing the context of application over stable storage also increases the overhead. In periodic interval checkpointing, sometimes processes takes checkpoints though it is not much useful. These unnecessary checkpoints increase the application's running time. We have proposed an algorithm which determines checkpointing interval dynamically, based on expected recovery time, to avoid unnecessary checkpoints. By eliminating unnecessary checkpoints, we can reduce running time of a process significantly.

List of Figures

2.1 An example of a consistent and inconsistent state. 11
2.2 Effects of Orphan Messages and Domino Effect 13
2.3 Lost Messages 13
2.4 State Before Livelock 14
2.5 Livelock Situation 14
2.6 Domino Effect 15
2.7 Z-path determination 20
2.8 Orphan process 24

3.1 Total number of messages used for coordination of a checkpoint 33

4.1 Page Table Entry of the i386 in the Linux Kernel 35
4.2 Execution without checkpointing 39
4.3 Execution with Incremental Checkpointing 40
4.4 Example Situation with Incremental Checkpoinitng 42
4.5 Comparison of average execution times for matrix multipication 46
4.6 Comparison of average execution times for quick sort 47

Chapter 1

INTRODUCTION

Distributed systems today are ubiquitous and enable many applications, including client-server systems, transaction processing, World Wide Web, and scientific computing, among many others. The vast computing potential of these systems is often hampered by their susceptibility to failures. Therefore, many techniques have been developed to add reliability and high availability to distributed systems.

1.1 DISTRIBUTED SYSTEM

A distributed system consists of a collection of independent computers that appears it's user as a single coherent system. These computers are communicated with each other by exchanging messages over a communication network. The machines in Distributed System offer their resources for collective computation.

These resources can be of various types such as storage devices, printers, computational nodes etc. The main objectives of the distributed system is to achieve high throughput for distributed application and to increase accessibility to resources not commonly availble to a single machine. A distributed application is composed of several processes running on different nodes of a distributed system and communicating through messages. Typical distributed system consist of following characterstics.

- *Multiple Nodes* :- A distributed system is composed of multiple independent nodes belonging to different computers, not merely multiple processors on the same computer.

- *Heterogeneity* : - The nodes in a distributed system may consist of machines having different architectures and possibly running different types of operating systems.

- *Message Passing* : - Processes on the different resource nodes may communicate using diverse networking protocols over different networking technologies. Therefore, the characteristics of the underlying communication links can be different. The nodes in most distributed systems are reachable from one another.

- *Concurrency* : - Each of the nodes in a distributed system provides independent functionality and operates concurrently with other nodes.

- *Decentralized Control* : - No single computer is necessarily responsible for configuration, management, or policy control for the whole distributed system. However,

some functionality may reside in a central node, or a set of nodes by necessity.

1.2 FAULT TOLERANCE IN DISTRIBUTED SYSTEM

Distributed systems are composed of multiple computing resources connected by co-communication links. Nodes in a Distributed System are susceptible to failures for many different reasons. Failure of nodes and links are assumed to be independent, larger the system, higher is its probability of failure. To provide fault tolerance, it is essential to understand the nature of the faults that occur in these systems.

There are mainly two kinds of faults: permanent and transient [15]. Permanent faults are caused by permanent damage to one or more components and transient faults are caused by changes in environmental conditions. Permanent faults can be rectified by repair or replacement of components. Transient faults remain for a short duration of time and are difficult to detect and deal with. Hence, it is necessary to provide fault tolerance particularly for transient failures in distributed system.

Distributed system should remain atleast partially available and functional even if the some of their nodes or communication links fail or misbehave. Without fault tolerance mechanisms, the system and applications running on it need to be restarted every time a failure occurs. Many of the distributed systems applications are long running, taking hours or even days in some cases to complete. If a fault occurs in the middle of a long running application, long hours of useful computation will be lost. Thus an application may take a long time to complete in the presence of such failures. In case of such failures the distributed system as a whole needs to be restored to an error free state, existing prior to failure. Fault tolerance techniques can allow applications to run to completion in the presence of faults with minimal disruption.

For recovering the system from this failures, it is better to start from a previous error free state. This fault tolerance can be achieved through the *checkpointing* [3]. Checkpoint is defined as a designated place in a program at which normal processing is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time. On the other hand, *rollback recovery* for an application is defined as the procedure for restarting the application process from a checkpoint stored in stable storage.

However, several issues with checkpoint and recovery technique arise in a distributed system which runs applications having multiple concurrent processes, communicating with each other via messages. Overheads of a checkpoint and recovery protocol include protocol overhead during checkpointing, protocol overhead during recovery and overhead for accessing stable storage. However, these overheads are different for different protocols. The checkpoint and recovery protocols proposed in literature can be broadly classified into the following four classes.

- **Independent Checkpointing Protocols**

  The protocols in this class allow processes to take local checkpoints independent of other processes in the distributed system. Such protocols are also referred to as uncoordinated or asynchronous checkpointing protocols. The fault-free runtime overhead is least for these kind of protocols because no coordination is needed between the processes to take checkpoints. During recovery, processes coordinate among themselves to determine a consistent global state. Therefore, recovery overhead is high. Due to bad placement of checkpoints over the communication pattern, the recovery protocol may require several rounds of coordination and rollbacks until a consistent global checkpoint is found. As a result a lot of useful computation may be lost and recovery overhead increases. Many of the local checkpoints taken may not be part of any recovery line and they are called useless checkpoints. However, processes may need to store all the local checkpoints since identifying which checkpoints are useless at runtime can be costly. As a result storage requirement is high for this kind of protocols.

- **Coordinated Checkpointing Protocols**

  In this class of protocols a process does not take local checkpoints independently, but synchronizes every checkpointing event with that of other processes, such that every checkpointing effort results in a consistent global checkpoint. These protocols are also known as synchronous checkpointing protocols. Checkpointing overhead is high due to the synchronization effort involved. However, recovery is fast, since processes only need to roll back to their latest local checkpoints. Storage space requirement is also minimum, because the protocols require that only the latest checkpoint be stored for recovery.

- **Communication Induced Checkpointing Protocols (CIC)**

  This class comprises of protocols which try to combine the positive aspects of both the independent and the coordinated checkpointing protocols. This class of protocols allows processes to take checkpoints independently, which are called basic checkpoints. In addition, the protocols use the communication pattern to force some extra checkpoints, called forced checkpoints. Forced checkpoints are taken to avoid creation of useless checkpoints in the system. Processes decide if a forced checkpoint should be taken or not, based on the information piggybacked on application messages by the sender. Hence processes may end up taking more number of local checkpoints in this case compared to that in the independent checkpointing protocols. Since the protocol prevents creation of useless checkpoints, processes need not keep all its checkpoints.

- **Log based Recovery Protocols**

  The previous three classes contain protocols with purely checkpoint based recovery. Log based recovery adopts a different strategy for recovery than the earlier three classes. Log based recovery protocols assume that processes follow the *piecewise deterministic model* (PWD) [29]. The execution of a process is modeled as a series of deterministic execution intervals terminated by non deterministic events, for example, the receive of a message. The first deterministic execution interval of a process begins with the start of the process. Every deterministic execution interval terminates with the first non deterministic event since its initiation. The same non deterministic event starts the next deterministic execution interval. If the non deterministic event e terminates an interval I and initiates an interval J, and if the event e can be replayed exactly at the end of interval J, then the execution of the interval J can be repeated. Recovery in this class of protocols depends on this principle. These protocols require that the content of the application messages and the information to replay them in order be stored and available during recovery.

  During recovery, the failed process is restarted and all the application messages received by it before failure is replayed. By the PWD model, the failed process can retrace the execution path and will eventually arrive at a state just before the failure, i.e., the process is rolled forward. To avoid a complete restart of the failed process, checkpointing is used in conjunction with message logging. The failed

process can then resume execution from a checkpointed state instead of a complete restart. This bounds the amount of rollback a process may suffer and recovery becomes faster. But the overhead of message logging during failure-free execution is a drawback for these protocols.

1.3 MOTIVATION

A distributed system employing checkpoint and recovery protocols provides improved throughput by virtue of its ability to recover a process from failure without complete restart of the application. In a failure prone environment, a checkpoint and recovery protocol saves recomputation and as a result reduces application completion time and increases throughput of the system. Better the performance of a checkpoint and recovery protocol, better is the throughput of a system.

The performance of a checkpoint and recovery protocol depends on the overheads it incurs against the amount of computation the protocol can save during recovery. The amount of computation a checkpoint and recovery protocol saves can be measured by the amount of reexecution that has to be done after a rollback. Smaller the reexecution, higher is the amount of computation saved.

The overheads incurred by a checkpoint and recovery protocol are dependent on many system and application characteristics of the checkpoint and recovery protocol. For example, if the fault frequency of a system is low, it makes sense to take checkpoints less frequently to reduce the checkpointing overhead. Similarly, if the application is communication-intensive and large loss of computation is undesirable, a coordinated checkpoint and recovery protocol may be preferred over an uncoordinated one to avoid possible cascaded rollbacks during recovery. For example, choosing a low checkpointing frequency, i.e., high checkpointing interval, may result in a large loss of computation in case a fault does happen.

Protocols involving communication among processes are affected by network traffic. Such a protocol incurs less overhead on a high speed network and high overhead in low speed ones. In addition, the application also affects the performance of a checkpoint and recovery protocol. For example in a message log-based recovery, all application messages are logged. So, message logging overhead is directly proportional to the number and size of application messages.

The total running time of an application is depend on the execution time of the application and the amount of checkpointing overhead that incurs with the application. We should minimize this checkpointing overhead. Checkpointing overhead is the combination of context saving overhead and coordination overhead. Storing the context of application over stable storage also increases the overhead.

1.4 OBJECTIVES OF THESIS

The main objective of our research work is to develop efficient checkpoitning algorithms. To address this broad objective, we identify the following goals:

- We wish to develop an efficient checkpointing algorithm in Distributed System which incurs the less coordination overhead and there by reducing the network traffic.
- Next, we want to propose an algorithm which determines checkpoint interval dynamically to get the more advantages over the periodic interval checkpointing.

1.5 ORGANIZATION OF THESIS

The rest of thesis work is organized as follows.

Chapter 2 presents discussion and literature survey on fault tolerance techniques in general and checkpoint and recovery protocols in particular.

Proposed coordinated checkpointing algorithm which reduces the chechpoint coordination overhead and we compared it with previously existed algorithms in chapter 3.

Chapter 4 presents dynamic interval deteminitation for incremental checkpoiting algorithm for reducing context overhead of checkpoints. First, we discussed some basic definitions of probability and calculus. Next, we derived some intermediate derivations.

Chapter 5 concludes the thesis with a summary of our contributions. We also briefly discuss the possible future extensions to our work.

Chapter 2

BACKGROUND AND RELATED WORK

Checkpoint is defined as a designated place in a program at which normal processing is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time. Checkpointing is the process of saving the status information. A checkpoint of a process is the information about the state of a process at some instant of time. Fault tolerance through checkpoint and recovery techniques includes taking checkpoint of an application process periodically and logging the checkpoint in a stable storage which is immune to failures. Checkpoint of an application process is the information about the state of the process and can be used to restart it from a state corresponding to the checkpoint. On the other hand, roll back recovery for an application is defined as the procedure for restarting the application process from a checkpoint stored in stable storage.

A checkpoint can be saved on either stable storage or the volatile storage of another process, depending on the failure scenarios to be tolerated. For long running scientific applications, checkpointing and rollback recovery can be used to minimize the total execution times in the presence of failures.

For an application involving a single process checkpoint and recovery is simple. In the event of a node failure, the application process is restarted from latest checkpoint, i.e. rolled back it's latest checkpoint. However, several issues with checkpointing and recovery arise in distributed system which runs applications having multiple concurrent processes, communicating with each other via messages.

2.1 ASPECTS OF CHECKPOINTING

Some of the aspects to be considered with checkpointing are frequency of checkpointing, contents of a checkpoint and methods of checkpointing [15].

2.1.1 Frequency of Checkpointing

A checkpointing algorithm executes in parallel with the underlying computation. Therefore, the overheads introduced due to checkpointing should be minimised. Checkpointing should enable a user to recover quickly and not to lose substantial computation in case of an error, which necessitates frequent checkpointing and consequently significant overhead. The number of checkpoints initiated should be such that the cost of information loss due to failure is small and the overhead due to checkpointing is not significant. This

depends on the failure probability and the importance of the computation. For example, in a transaction processing system where every transaction is important and information loss is not permitted, a checkpoint may be taken after every transaction, increasing the checkpointing overhead significantly.

2.1.2 Contents of a Checkpoint

The state of a process has to be saved in stable storage so that the process can be restarted in case of an error. The state context includes code, data and stack segments along with the environment and the register contents. Environment has the information about the various files currently in use and the file pointers. In case of message passing systems, environment variables include those messages which are sent and not yet received.

2.1.3 Methods of Checkpointing

The methodology used for checkpointing depends on the architecture of the system. Methods used in multiprocessor systems should incorporate explicit coordination unlike uniprocessor systems. In a message passing system, the messages should be monitored and if necessary saved as part of the global context. The reason is that, messages introduce dependencies among the processors. On the other hand, a shared memory system communicates through shared variables which introduce dependency among the nodes and thus, at the time of checkpointing, the memory has to be in a consistent state to obtain a set of concurrent checkpoints.

2.2 OVERHEADS OF CHECKPOINTING

During a failure free run, every global checkpoint incurs coordination overhead and context saving overhead in a distributed system. We define them as follows.

- **Coordination Overhead**

  In a distributed system, coordination among processes is needed to obtain a consistent global state. Special messages and piggy-backed information with regular messages are used to obtain coordination among processes. Coordination overhead is due to these special messages and piggy-backed information.

- **Context Saving Overhead**

  The time taken to save the global context of a computation is defined as the context saving overhead. Overhead for checkpoint storage in stable storage contributes a major part of the overhead of checkpoint and recovery protocols. This overhead is proportional to the size of the context.

2.3 CONSISTENT SYSTEM STATES

In distributed system, a computation node can take checkpoints of its local processes only and such checkpoints are called *local checkpoints* [7]. A *global checkpoint* of a distributed system is defined as set of local checkpoints, one from each of its processes in the system. After recovery the system must be in a *consistent state*. A global state of a message passing system is a collection of the individual states of all participating processes and of the states of the communication channels. Intuitively, a consistent global state is one that may occur during a failure free, correct execution of a distributed computation. More precisely, a consistent system state is one in which if a processs state reflects a message receipt, then the state of the corresponding sender reflects sending that message.

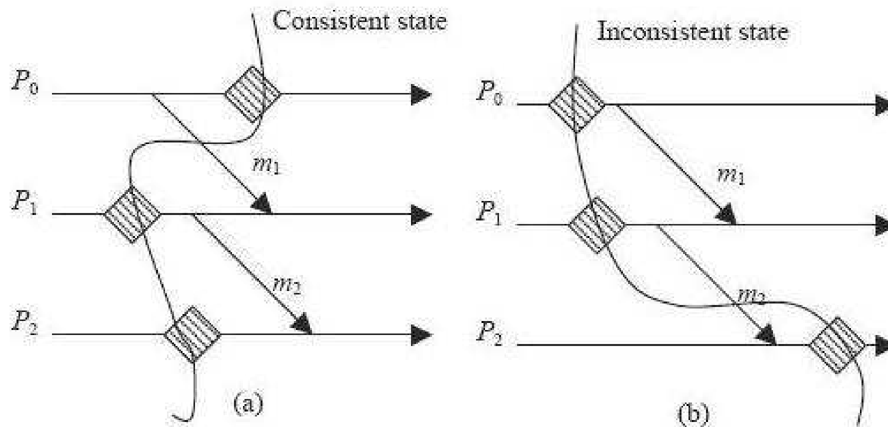


Figure 2.1: An example of a consistent and inconsistent state.

For example, Fig 2.1 shows two examples of global states, a consistent state in Fig 2.1(a) and an inconsistent state in Fig 2.1(b). Note that the consistent state in Fig 2.1(a) shows message m_1 to have been sent but not yet received. This state is

consistent, because it represents a situation in which the message has left the sender and is still travelling across the network. On the other hand, the state in Fig 2.1(b) is inconsistent because process P_2 is shown to have received m_2 but the state of process P_1 does not reflect sending it. Such a state is impossible in any failure free, correct computation. Inconsistent states occur because of failures. For example, the situation shown in part (b) of Figure 2.1 may occur if process P_1 fails after sending message m_2 to P_2 and then restarts at the state shown in the figure. A fundamental goal of any rollback recovery protocol is to bring the system into a consistent state when inconsistencies occur because of a failure. The reconstructed consistent state is not necessarily one that has occurred before the failure. It is sufficient that the reconstructed state be one that could have occurred before the failure in a failure free, correct execution.

2.4 ISSUES IN CHECKPOINTING

In concurrent systems, several processes cooperate by exchanging information to accomplish task. The information exchanges through the message passing. In such system, if one of the cooperating process fails and resumes execution from a recovery point, then the effects it has caused at other processes due to the information it has exchanged with them after establishing the recovery point will have to be undone. To undo the effects caused by a failed process at an active process, the active process must also rollback to an earlier state. Rolling back processes in concurrent system is more difficult than in the case of a single process. The following discussion illustrates how the rolling back of processes can cause further problems [20].

2.4.1 Orphan Messages and Domino Effect

Consider the three processes X, Y and Z are running concurrently in Fig 2.2. The parallel lines are showing the executions of the processes. These processes are communicated through exchange of messages. Each symbol '[' marks a recovery point to which process can be rolled back in the event of a failure.

If the process X is to be rolled back, it can be rolled back to the recovery point x_3 without affecting any other process. Suppose that Y fails after sending message m and is rolled back to y_2. In this case, the receipt of m is recorded in x_3, but the sending of m is not recorded in y_2. Now we have a situation where X has received message m from Y,

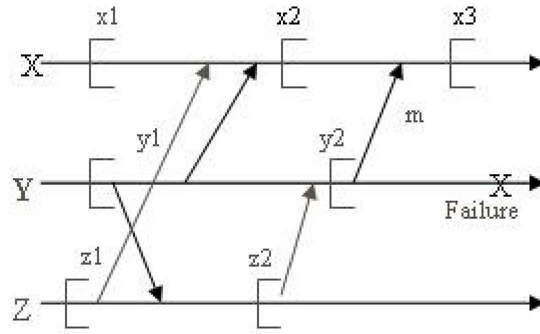


Figure 2.2: Effects of Orphan Messages and Domino Effect

but Y has no record of sending it, which corresponds to an inconsistent state. Under such circumstances, m is reffered to as an *orphan* message and process X must also roll back. X must roll back because Y interacted with X after establishing its recovery point y_2. When Y is rolled back to y_2, the event that is responsible for the interaction is undone. Therefore, all the effects at X caused by the interaction must also be undone. This can be achievied by rolling back X to recovery point x_2. Likewise, it can be seen that, if Z is rolled back, all three processes must rollback to their very first recovery points, namely, x_1, y_1, z_1. This effect, where rolling back one process causes one or more processes to roll back, is known as the $domino\ effect$ and orphan messages are the cause.

2.4.2 Lost Messages

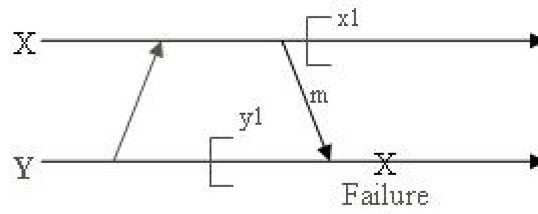


Figure 2.3: Lost Messages

Suppose that checkpoints x_1 and y_1 in Fig 2.3 are chosen as the recovery points for processes X and Y, respectively. In this case, the event that sent message m is recorded in x_1, while the event of its receipt at Y is not recorded in y_1. If Y fails after receiving

message m the system is restored to state x_1, y_1, in which message m is *lost* as process X is past the point where it sends message m. This condition can also arise if m is lost in the communication channel and processes X and Y are in state x_1 and y_1, respectively.

2.4.3 Problem of Livelocks

In rollback recovery, livelock is a situation in which a single failure can cause an infinite number of rollbacks, preventing the system from making process. A livelock situation in a distributed system is shown in Fig 2.4.

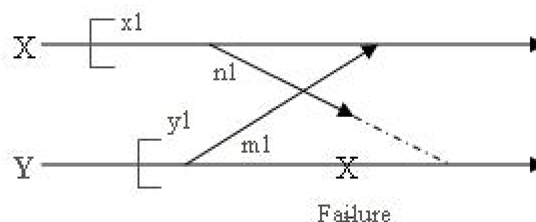


Figure 2.4: State Before Livelock

Fig 2.4 illustrates the activity of two processes X and Y until the failure of Y. Process Y fails before receiving message n_1, sent by X. When Y rolls back to y_1, there is no record of sending message $m1$, hence X must rollback to x_1.

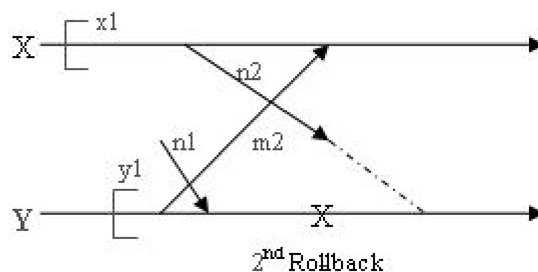


Figure 2.5: Livelock Situation

When process Y recovers, it sends out m_2 and receives n_1 (Fig 2.5). Process X, after resuming from x_1, sends n_2 and receives m_2. However, because X rolled back, there is no record of sending n_1 and hence Y has to roll back for the second time. This forces

X to rollback too, as it has received m_2 and there is no record of sending m_2 at Y. This situation can repeat indefinitely, preventing the system from making any progress.

2.5 DIFFERENT TYPES OF CHECKPOINTING PROTOCOLS

Several checkpointing and recovery protocols are available in the literature [11]. These protocols are classified into following categories. (i) Independent checkpointing protocols (ii) Coordinated checkpointing protocols (iii) Communication induced checkpointing protocols (iv) Log based recovery protocols

2.5.1 Independent Checkpointing Protocols

The protocols in this class allow to take local checkpoints independent of other processes in the distributed system. Such protocols are also reffered as uncoordinated or asynchronous checkpointing protocols. The fault free run time ocerhead is least for these kind of protocols because no coordination is needed between the processes to take checkpoints. During recovery, processes coordinate among themselves to determine a consistent global state. Therefore, recovery overhead is high. Due to bad placement of checkpoints over the communication pattern, the recovery protocol may require several rounds of coordination and rollbacks until a consistent global checkpoint is found. As a result a lot of useful computation may be lost and recovery overhead increases.

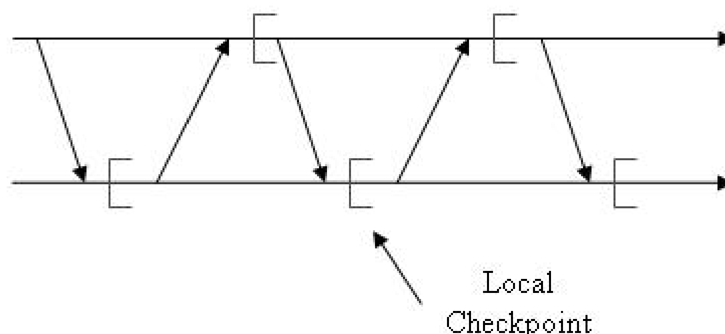


Figure 2.6: Domino Effect

Many of the local checkpoints taken may not be part of any recovery line and they are

called *useless* checkpoints. However, processes may need to store all the local checkpoints since identifying which checkpoints are useless at runtime is costly. As a result storage requirement is high for this kind of protocols. There is also a possibility of domino effect which may cause a loss of large amount of useful computation [26]. Fig 2.6 shows a checkpoint and communication pattern involving two processes, which can be affected by domino effect. It can be seen from the figure that all the possible global checkpoints are inconsistent and therefore all local checkpoints are useless checkpoints. If a fault occurs, the recovering process will force the other process to roll back to its previous checkpoint, until both the processes are rolled back to their initial state.

In order to determine a recovery line during recovery, processes record their dependencies with checkpoints of other processes during failure free execution. The straightforward way to keep track of such information is by using a set of messages counters, one for each of the processes with which it communicates. When a process sends out an application message, it increments the counter value corresponding to the receiver process, and tags the value as an identification number to the message. The processes also maintain records of the highest numbered message received from each of its senders. Processes store both these send and receive information along with its checkpoints and this is used to determine inconsistent checkpoints during recovery.

A checkpoint of a process P is inconsistent with that of a process Q if the highest message id received from the process Q recorded in P's checkpoint is higher than the send counter value corresponding to P as recorded in Q's checkpoint. Then process P has to roll back again. This may lead to several rounds of coordination and cascaded rollbacks until a consistent global checkpoint is found. These dependency tracking protocols add some overhead during fault free execution. In case of a fault, all the processes have to coordinate among themselves to decide upon a recovery line to which they will roll back. Therefore, recovery overhead is high in most cases. This approach is very reasonable if faults are rare in the system under consideration. There is also a possibility of domino effect during recovery. This class of protocols does not inherently support output commit.

2.5.2 Coordinated Checkpointing Protocols

In this class of protocols a process does not take local checkpoints independently, but synchronizes every checkpointing event with that of other processes, such that ev-

ery checkpointing effort results in a consistent global checkpoint. These protocols are also known as synchronous checkpointing protocols. This class of protocols ensures that whenever a process takes a local checkpoint, all other processes in the system also take their respective local checkpoints. As a result every local checkpointing effort translates into a global checkpointing activity. They are free from domino effect. Storage space requirement is minimum for these protocols, since they require that only the latest checkpoint be stored for recovery. All the checkpoints taken with successful coordination are useful, i.e., the recovery line steadily progresses with every checkpoint. All the latest local checkpoints are part of a consistent global checkpoint and therefore recovery time is lower compared to independent protocols. However, due to the effort of synchronization involved in every checkpointing activity, checkpointing overhead is high.

Coordinated checkpointing protocols can be either blocking or non blocking.

- **Blocking protocols**

  A straightforward approach to coordinated checkpointing is to block inter process communication until the checkpointing protocol ompletes [8, 31] . The protocol is initiated by a coordinator. The coordinator sends a request to all processes asking them to checkpoint. On the receipt of the request the process blocks normal execution, takes a tentative checkpoint and sends an acknowledgment message to the coordinator. On receipt of the acknowledgment messages from all the processes, the coordinator sends a message indicating the end of the protocol. On receipt of this message, all processes make their tentative checkpoints permanent, remove old permanent checkpoints and resume normal execution. Acharya and Badrinath [1] have devised a blocking coordinated checkpointing protocol on the assumption of causal order message delivery by the underlying communication system. Koo and Toueg [16] proposed a blocking coordinated checkpointing protocol which allows failure during checkpointing while storing only two local checkpoints per process. They showed that storage of two checkpoints is the minimal requirement to tolerate failure during checkpointing. Another important feature of their protocol is that only a subset of the processes need to take checkpoint.

- **Non blocking protocols**

  The problem with blocking coordinated checkpointing protocols is that the processes are not allowed to execute until the coordination is complete. Hence check-

pointing overhead is high. Chandy and Lamport's [7] distributed snapshot protocol provides a non blocking coordinated checkpointing protocol over reliable FIFO links. They use the property of the underlying link to achieve coordinated checkpointing without the explicit two round coordination protocol. The idea is to use a special message, called a *marker message*, which is a checkpoint request message carrying the checkpointing interval number. The protocol is similar to the flooding protocols used in distributed systems. The process which first initiates coordination blocks the local process, takes a local checkpoint, sends a marker message to all its neighboring processes and then unblocks the local process. The receiver of a marker message, if has not already received a similar message, follows the same procedure as that of the coordinator and forwards the marker message to all its neighbors. Lai and Yang [17] relaxed the FIFO constraint by piggybacking the marker on every post checkpoint message. The same affect can be obtained by marking local checkpoints by a sequence number, called checkpoint index and piggybacking the current checkpoint index value on every application message [12, 27].

Similar to blocking coordinated checkpointing protocols, attempts have been made to construct non blocking coordinated checkpointing protocols which require that a minimal number of processes take checkpoints. The coordination protocols discussed above involve all the processes in the system and therefore raises concern for scalability of the protocols. Prakash and Singhal [25] observed that only those processes which have communicated since the last checkpoint require to checkpoint again. A minimal number of processes, only those whose present states are causally dependent on the current state of the coordinator, need to participate. They proposed a non blocking coordinated checkpointing protocol where every process keeps track of the processes which are causally dependent on its present state and the protocol involves only those processes which are causally related to the coordinator.

However, Cao and Singhal found a flaw in Prakash and Singhal's protocol and went on to prove the impossibility of a non blocking coordinated protocol where a minimal number of processes participate [5] . Cao and Singhal also proposed a non blocking coordinated checkpointing protocol [6] , using mutable checkpoints,

which reduces the number of checkpoints to be stored in stable storage. Mutable checkpoints can be stored in the volatile memory and are converted into permanent checkpoints and stored in stable storage only when a new recovery line is developed.

Another approach to non blocking coordinated checkpointing protocol is to avoid explicit coordination by message passing and use synchronous clocks to achieve implicit coordination. In modern distributed systems many applications require clocks of the processors to be approximately synchronized. Many distributed systems run clock synchronization protocol at the background to keep their clock differences within some guaranteed bound [18] . Such loosely coupled synchronized clocks can facilitate checkpointing effort without explicit coordination . A process takes a local checkpoint and blocks all receives for a period which is equal to the maximum deviation between clocks plus the maximum time to detect a failure in the system. It can be shown that all its local checkpoints of processes form a global recovery line.

2.5.3 Communication Induced Checkpointing Protocols (CIC)

This class comprises of protocols which try to combine the advantages of both the independent and the coordinated checkpointing protocols. This kind of protocols take two types of checkpoints, namely basic and forced checkpoints. Basic checkpoints can be taken independently by the processes, while forced checkpoints are extra checkpoints forced by the communication pattern to avoid the creation of useless checkpoints in other processes. These protocols ensure that every checkpoint taken by processes is in some recovery line. Since local checkpoints are taken independently these protocols suffer less overhead for checkpointing, and yet avoid domino effect. But these protocols may end up taking more number of checkpoints compared to that in independent checkpointing protocols. The failed process can determine the recovery line without any coordination with other processes. Other processes need to roll back to the recovery line sent by the failed process. Therefore recovery is simple.

The protocols in this class piggyback protocol-specific information on every application messages. The receiver process then analyzes this information to decide whether any forced checkpoint is required or not. If so, the process first takes a checkpoint and then delivers the message to the application. Informally, the decision is based on whether the

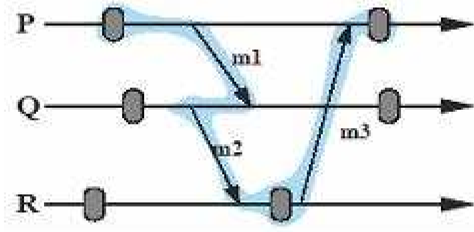


Figure 2.7: Z-path determination

checkpoint and communication pattern will create any useless checkpoints in the system. If there is such a possibility, a forced checkpoint is taken to break the pattern. The decision is based on the notion of a $Z-cycle$ and a $Z-path$, based on the *zigzag path* formulation [7]. A $Z-path$ is the same as a zigzag-path. A Z-cycle is a Z-path that begins and ends in the same checkpoint interval (Fig 2.7). CIC protocols can be broadly sub-divided into two classes, $index-based\ coordination\ protocols$ and $model-based\ checkpointing\ protocols$.

In index based coordination checkpointing protocol, a process takes both basic and forced checkpoints, and all local checkpoints of a process are indexed by a monotonically increasing value. The index value is piggybacked on all application messages. When a process receives an application message, it checks whether the piggybacked index value is higher than its own. If so, then it updates its own index value to the piggybacked index value and takes a forced checkpoint. It then delivers the message to the application process. The protocol ensures that local checkpoints in different processes having the same index value form a recovery line. A more sophisticated protocol where processes transmit more information on application messages to reduce the number of forced checkpoints was proposed in [13].

A model based checkpointing protocol defines a model of checkpoint and communication pattern which contains no useless checkpoints. For example, a model of checkpoint and communication pattern can be defined as the one which does not have any Z-cycle. A checkpoint and communication pattern, free of Z-cycle, does not contain any useless checkpoints. Therefore, a protocol which enforces such a model, ensures that no useless checkpoint is generated in the system. When a process detects a possibility of violation of any constraints put forward by the model, it takes a forced checkpoint. All decisions

are taken locally with the help of the information gathered from other processes through piggybacked values. Since the processes take independent decisions, the possibility of violation of the model can be detected by multiple processes at the same time and all of them may take preventive actions. Therefore, processes may end up taking more checkpoints than actually required.

Wang [32] has classified these protocols as follows:.

- $NRAS$ or $No - Receive - After - Sendprotocol$, where a checkpoint has to be taken before the first message received after a message has been sent.
- CAS or $Checkpoint - After - Send$, where a checkpoint is forced after every send event.
- CBR or $Checkpoint - Before - Receive$, where a checkpoint is forced before all receive events.
- $CASBR$ or $Checkpoint - After - Send\ and\ Before - Receive$, is a combination of CAS and CBR protocols. Here, checkpoints are taken after a message is sent and also before a message is received.
- $FDAS$ or $Fixed - Dependency - After - Send$, uses checkpoint sequence number (csn) to track dependency among processes. Every message sent carries this number. If a received message carries a csn value higher than the local csn value of the receiver, the receiver process first updates its local csn value to the received csn value, then takes a forced checkpoint, and then delivers the message to the application process.
- FDI or $Fixed\ Dependency\ Interval$, protocols force checkpoints before dependency of a process changes due to some receive event. Manivannan and Singhal have proposed such a protocol [21].

Helary et. al. showed that since the checkpoint index number in index based coordination protocols always increases along a Z-path, no Z-cycle can be formed. They have also proved that index based checkpointing is a form of model based checkpointing, specifically belonging to the FDAS class.

2.5.4 Log based Recovery Protocol

The previous three classes contain protocols with purely checkpoint based recovery. Log based recovery adopts a different strategy for recovery than the earlier three classes. These protocols log all application messages. When a failed process is restarted, the logged messages are replayed to it exactly at the same instances as they were received before its failure. The PWD model ensures that the same execution path is retraced by the process in this technique, i.e., the process can be rolled forward. The execution of a process is modeled as a series of deterministic execution intervals terminated by non deterministic events, for example, the receive of a message. The first deterministic execution interval of a process begins with the start of the process. Every deterministic execution interval terminates with the first non deterministic event since its initiation. The same non deterministic event starts the next deterministic execution interval.

If the non deterministic event e terminates an interval I and initiates an interval J and if the event e can be replayed exactly at the end of interval I, then the execution of the interval J can be repeated. Recovery in this class of protocols depends on this principle. These protocols require that the content of the application messages and the information to replay them in order be stored and available during recovery.

During recovery the failed process is restarted and all the application messages received by it before failure is replayed. By the PWD model, the failed process can retrace the execution path and will eventually arrive at a state just before the failure, i.e., the process is rolled forward. To avoid a complete restart of the failed process, checkpointing is used in conjunction with message logging. The failed process can then resume execution from a checkpointed state instead of a complete restart. This bounds the amount of rollback a process may suffer and recovery becomes faster. But the overhead of message logging during failure free execution is a drawback for these protocols. Checkpointing is used in conjunction with log based recovery techniques to bound the amount of rollback. Three classes of message log based recovery protocols have been proposed based on three types of message logging techniques. The problem of output commit is inherently handled by protocols in this class.

- **Pessimistic Message Logging Protocols**

  This kind of protocols takes a conservative view of failure in that a fault can occur immediately after a nondeterministic event. Therefore, they ensure that all non

deterministic events are logged before it can affect the state of a process. All receive messages are logged in the stable storage before it is delivered to the application. Therefore this technique is also referred to as *synchronous logging*. The process which logs a message gets blocked until the message is logged in some stable storage. In this class of protocols, only the failed process needs to recover and the recovering process can recover upto the last state before failure. No orphan message is created.

This class of protocols always support output commit since the whole history of messages are always logged in the stable storage. Since only the failed process requires to recover, independent checkpointing protocols can be used in conjunction with this class of protocols. So, a process can choose any checkpointing frequency to limit its rollback. Garbage collection is simple because all checkpoints and message log of events prior to the most recent checkpoint can be purged. The overhead of synchronous logging can be reduced by special hardware or by bounding the number of tolerable failures.

- **Optimistic Message Logging Protocols**

  Unlike pessimistic message logging protocols, protocols in this class log messages in the volatile memory and therefore message logging overhead is lower. Message logs in volatile memory are asynchronously flushed to stable storage periodically, without blocking the application processes [28, 29, 33] . This class of protocols optimistically assumes that failure will not occur until the volatile log is flushed into the stable store which allows the failed process to recover to the last state before its failure similar to pessimistic protocols.

  However, when a process fails, all message logs stored in its volatile memory are lost and the process can not roll forward to its latest state before failure. If the recovering process had sent messages in the interval between its recovered state and the state before its failure, then such messages become orphan messages. Therefore, the recovering process may create orphan processes in the system.

  Consider a situation depicted in Fig 2.8. Let the message m_1 be logged by the process P in its volatile log. The process P fails after sending message m_2 and rolls

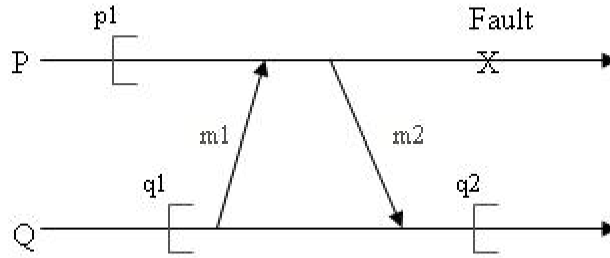


Figure 2.8: Orphan process

back to p_1. Since m_1 is not recorded, the process can not roll forward to the state before it receives m_1. So m_2 becomes an orphan message and Q becomes an orphan process. The possibility of creation of orphan processes complicates the recovery and garbage collection processes. Also, processes must be blocked in order to ensure that all volatile message logs are flushed to stable storage to ensure output commit.

To perform rollbacks correctly, optimistic logging protocols track causal dependency during failure free run. During recovery, this information is used to calculate a global state with no orphan processes. The recovery process is similar to that in independent checkpointing protocols. Multiple processes may need to roll back to avoid the creation of an orphan process. Processes need to store multiple checkpoints.

In Fig 2.8 the process Q is forced to roll back to q_1 in order to bring the system to a consistent state. Moreover, since at failure multiple processes may need to roll back, output commit generally requires the coordination of multiple processes. Recovery can be either *synchronous* or *asynchronous*.

–Synchronous Recovery

In this kind of protocols every process maintains information about causal dependency between processes, developed due to application message transmissions. This information is then used by the recovering process to determine the processes which need to roll back to avoid the creation of orphan processes.

–Asynchronous Recovery

In contrast to synchronous recovery, asynchronous recovery requires that processes

participate in multiple rounds of coordination to identify the processes which need to recover and a global state where they can rollback [29]. The process is similar to the recovery procedure for independent checkpointing protocol. It was shown that an exponential number of message may be exchanged to determine such a global state. The advantage is that, unlike synchronous recovery protocols, these protocols use very small piggybacked information and need not do elaborate book-keeping of dependency informations, and as a result have less failure free overhead. But recovery may become costly.

- **Causal Message Logging Protocols**

  This class of protocols uses the application communication pattern to ensure that whenever a process P sends a message m, P is recoverable atleast upto a state beyond the send event e^p_m, so that even if P fails, the receiver of m need not roll back. The protocol satisfies this condition by ensuring that the messages required to roll forward P beyond the state e^p_m are either stored in stable storage or are available in the volatile memory of some non failed processes. Casual logging has the advantage of the low failure free overhead of optimistic logging, while retaining most of the advantages of the recovery of pessimistic logging [2, 10]. However, recovery and garbage collection procedures are complex.

2.6 REDUCTION OF CHECKPOINTING OVERHEAD

Overhead of saving checkpoint over stable storage contributes a major part of the overhead of checkpoint and recovery protocols. Studies of performance evaluation of checkpointing protocols indicate that transfer of checkpointed information to stable storage incurs large overhead . The simplest way to store a checkpoint is to suspend the process until the checkpoint is written on stable storage. However several techniques are proposed to reduce this overhead.

- **Concurrent Checkpointing**

  Concurrent checkpointing is a hardware assisted method which allows checkpoints to be stored in a stable storage while the process is in execution. In this technique the address space of a process is protected from update [23] . If a process attempts

to write in a page, a fault is signaled. The page is then copied into a different unprotected buffer.

- **Pagelevel Incremental Checkpointing(PIC)**

  Pagelevel incremental checkpointing [24] only writes those pages which have been modified since the last checkpoint. It can be implemented using memory protection hardware or through implementation of dirty bit in operating systems . An alternative method is to directly compare the current checkpoint with the previous checkpoint, calculate the difference and finally store the difference only. However, the overhead of computation of the difference may offset the gain achieved from incremental storage.

- **System-level vs Application-level Checkpointing**

  Checkpoint of an application process can either be captured with system level checkpointing where the support for checkpointing is built in the kernel [4]. Alternatively, the application can define and store its state with the help of libraries linked with it. System level implementations are more powerful, because they can capture user level data and also have access to kernel level data structures which support applications. But the checkpoints are not portable between different systems. On the other hand application level checkpoints are portable. But this kind of checkpointing cannot access the kernel level data structure, such as open file descriptors, message buffers, etc. An advantage of application level checkpointing is that the application can choose a point in the program where checkpoint size is small and store the state.

- **Diskless Checkpointing**

  This method [22] was proposed by Plank which uses volatile memory as stable storage. The method uses coordinated checkpointing protocol as the basic checkpointing service. All the local checkpoints are collected and encoded in a processor (called the checkpoint processor) and a copy of it is stored in several volatile storages. To tolerate f number of failures the encoded global checkpoint is stored in $f + 1$ volatile memories. Therefore, even after f concurrent failures, the global state is still recoverable.

With diskless checkpointing, the processor saves its state in memory, rather than on disk. In its simplest form, diskless checkpointing requires an in-memory copy

of the address space and registers. If a rollback is required, the contents of the address space and registers are restored from the in-memory checkpoint. Note that this checkpoint will not tolerate the failure of the application processor itself; it simply enables the processor to roll back to the most recent checkpoint if another processor fails. If a rollback is required, the contents of the address space and registers are restored from the in-memory checkpoint. Note that this checkpoint will not tolerate the failure of the application processor itself; it simply enables the processor to rollback to the most recent checkpoint if another processor fails.

- **Forked Checkpointnig**

  The application protocols that are discussed till now, are halting their execution while taking checkpoints. If application takes more number of checkpoints then its execution time will be increases because of its halting execution. This problem can be overcome by forked checkpointing. Here, it makes a copy of the program's data space and to use an asynchronously executing thread of control to write the checkpoint file. This is called " main-memory checkpointing" [14] and improves checkpoint overhead if there is enough physical memory to hold the checkpoint, as the saving of the checkpoint to disk is overlapped with the execution of the application.

  The Unix $fork()$ primitive provides exactly the mechanism needed to implement main memory checkpointing.

- **Checkpoint Compression**

  With checkpoint compression, a standard compression algorithm like LZW [30] is used to shrink the size of the checkpoint. While this may be successful at reducing checkpoint size, it only improves the overhead of checkpointing if the speed of compression is faster than the speed of disk writes and if the checkpoint is significantly compressed. For uniprocessor checkpointing this is not the case. Compression has only been shown to be effective in parallel systems with disk contention.

A Coordination checkpoint algorithm has proposed in next chapter which reduces the number of messges that are needed for coordination, for taking a checkpoint.

Chapter 3

A CHECKPOINTING ALGORITHM FOR REDUCING COORDINATION OVERHEAD

To overcome the problems that are discussed in Sec 2.4, R. Koo and S. Toueg has proposed an algorithm [16]. The algorithm [16], which is caused to take minimal number of processes, is forced to take checkpoints. This algorithm overcomes these problems by dividing the entire thing into two phases and having some assumptions. In first phase, an initiating process takes a tentative checkpoint and sends the checkpoint request to all the processes that are in its cohort. The processes that are received checkpoint request message, takes a tentative checkpoint if it is intersted in taking checkpoint and send "yes" as its response to the initiator. The processes that are not interested in checkpointing will send "no" as its response to the initiator.

In second phase, if the initiator receives "yes" from all the processes in its cohort then it will make the decision as " make tentative checkpoint as permanent" else it will make the decision as "discard the tentative checkpoint". After making the decision, initiator informs this decision to all the processes in it's cohort. After receiving the decision from the initiator , the processes will act accordingly.

3.1 PROPOSED ALGORITHM

The algorithm in [16] takes three messages per processes for a single checkpoint coordination. One checkpoint request message, one response message and final ddecision message. Total three messages per process are needed for a single checkpoint coordination. If the number of processes in the system increases then the total number of messages for coordination will be increases. If the number of messages in the network increases then it increases congestion in the network. This is not desirable. These unnecessary messages are increasing congestion in the network. We should try to reduce the network congestion. This can be achieved by reducing the unnecessary messages. The algorithm in [16] is taking three messages per process for coordinating a single checkpoint. We should reduce the number of messages that are needed for coordination. This can be achieved by the following algorithm. The proposed algorithm is having the same assumptions that of the algorithm proposed by R. Koo and S. Toueg [16].

Messages that are not sent by the checkpoint algorithm are system messages. Every system message m contains a label $m.l$. Each process appends outgoing system messages with monotonically increasing labels.

The following notations are used in the algorithm.

L: largest label; S: smallest label

Let m be the last message X received from Y after X's last permanent checkpoint.
$Last_msg_recd_X[Y]$ = m.l, if m exists. Otherwise, it is set to S.

Let m be the first message X sent to Y after checkpointing at X (permanent or tentative).
$First_msg_sent_X[Y]$ = m.l, if exists. Otherwise, set to L.

For a checkpointing request to Y, X sends $Last_msg_recd_X[Y]$. Y takes a temporary checkpoint iff $Last_msg_recd_X[Y] \geq First_msg_sent_Y[X] >$ S.
i.e., X has received 1 or more messages after checkpointing by Y and hence Y should take checkpoint.

$Ckpt_cohort_X$ is the set containing all the processes from which X has received messages after it has taken its last checkpoint.

Each process will maintain a variable $Will_to_take_ckpt$.

Algorithm:

Initial State at all processes:

$First_msg_sent_X[Y]$ = S.

$Will_to_take_ckpt_X$= "yes" **if** X is willing;

"no" Otherwise

At initiator process X_i :

for all Y in $ckpt__cohort_X$ **do**

Send $Take__a__tentative__ckpt(X_i, Last__msg__recd_X[Y])$ message.

Wait for particular time period.

if any process replied "no" **then** for all Y in $ckpt__cohort_X$ **do**

Send $Undo__tentative__ckpt$.

else

Send $Make__tentative__ckpt__permanent$.

At all processes Y:

Upon receiving $Take__a__tentative__ckpt$ message from X **do**

if $Will__to__take__ckpt_Y =$ "yes" AND $Last__msg__recd_X[Y] \geq First__msg__sent_Y[X]$

$>$ S

Take a tentative checkpoint.

for all processes Z in $ckpt__cohort_Y$ **do**

Send $Take__a__tentative__ckpt(Y, Last__msg__recd_Y[Z])$ message.

Wait for particular time period .

if any process Z replied "no" **then**

$Will__to__take__ckpt_Y =$ "no" and Send (Y, $Will__to__take__ckpt_Y$) to X.

Upon receiving $Make__tentative__ckpt__permanent$ message **do**

Make tentative checkpoint permanent

for all processes Z in $ckpt__cohort_Y$ **do**

Send $Make__tentative__ckpt__permanent$ message.

Upon receiving $Undo__tentative__ckpt$ message **do**

Undo tentative checkpoint.

for all processes Z in $ckpt__cohort_Y$ **do**

Send $Undo__tentative__ckpt$ message.

The proposed algorithm is reducing the number of messages that are needed for a checkpoint coordination as like this. The algorithm in [16] takes three messages per process (request message, reply message, final decision message) for coordinating the checkpoint process. But, the proposed algorithm will take only two messages for a successful checkpoint. One message is for request and one message is for final decision message. Here, there are no reply messages because the processes that are interested in taking checkpoint will not send any reply message and these processes will directly take tentative checkpoint. After receiving the final decision message, they will act according to the final decision message. But, according to our algorithm, only two messages per process are sufficient for checkpoint coordination.

By decreasing one message per process, we can reduce checkpoint overhead and the network congestion significantly. Reply message will come into existence, only when a process is not interested in taking checkpoint. In an average case, if the half of the processes send replies, then the number of messages needed for checkpoint coordination will be less than three messages.

3.2 PERFORMANCE EVALUATION

In this section, we present the experimental results of the proposed algorithm. Here, we have calculated the total number of messages that are needed for single checkpoint coordination and compared with the total number of messages that are needed with the algorithm [16]. In this case, not all the processes are sending reply messages. Based on the number of processes sending reply messages, we had calculated the total number of messages that are needed for coordination of a checkpoint. Here, we eliminated reply messages by considering the assumptions that are discussed in [16]. Proposed algorithm is using only two messages per process for successful checkpoint and the total number of messages that are needed for checkpoint coordination is also less. In Fig 3.1, the upper line gives the values that are obtained by using the algorithm[[16]. Below the upperline, remaining lines represents the values that are obtained by our proposed algorithm with 60%, 30% and none of the processes has send reply messages.

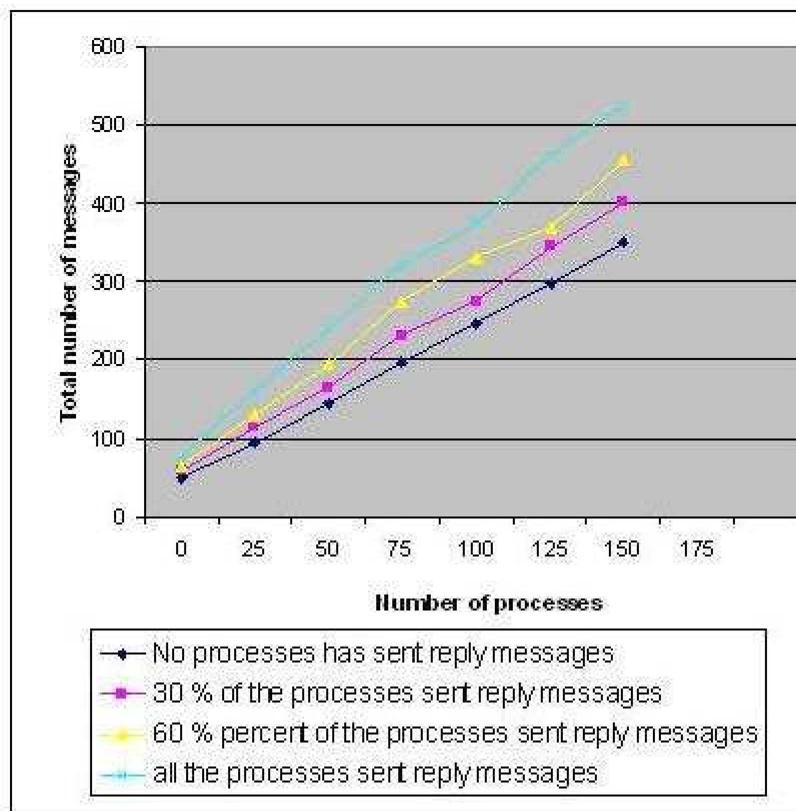


Figure 3.1: Total number of messages used for coordination of a checkpoint

Chapter 4

DYNAMIC INTERVAL DETERMINATION FOR PIC

Checkpointing is an effective mechanism to prevent a process from restarting the execution from the initial state in case of system failures. But there is checkpointing overhead to save memory state of a process. Saving the state means it incurs with the stable storage. The speed of writing data to a secondary storage is much slower than the execution of the process. It is very difficult to fill that gap. To reduce the running time of a process, we have to reduce the disk writing overhead.

When a checkpoint is taken, only the portion of the checkpoint that has changed since the previous checkpoint need to be saved. The unchanged portion can be restored from previous checkpoints. Several techniques have been devised and implemented to minimize the checkpointing overhead. These can be divided into two groups. One is the latency hiding optimization techniques such as diskless checkpointing, forked checkpointing and compression checkpointing which attempt to reduce or hide the disk writing overhead. The other is the size reduction techniques such as memory exclusion checkpointing and incremental checkpointing [24] which attempt to minimize the amount of data that gets stored per checkpoint. An important point to note with respect to size reduction is that the large amounts of read only memory or unmodified memory pages are identified and excluded from checkpoints. Among these techniques, incremental checkpointing is widely used in practically. In incremental checkpointing, the large amounts of read-only memory or unmodified memory pages are identified and excluded from checkpoints.

These modified pages or dirty pages information can be extracted by using the dirty bit. This dirty bit information will be available in $page\ fault$ routine in linux kernel. The page table entry of i386 architechture as shown in Fig 4.1. For implementing this we were used D bit, R/W bit and Available bits in the figure.

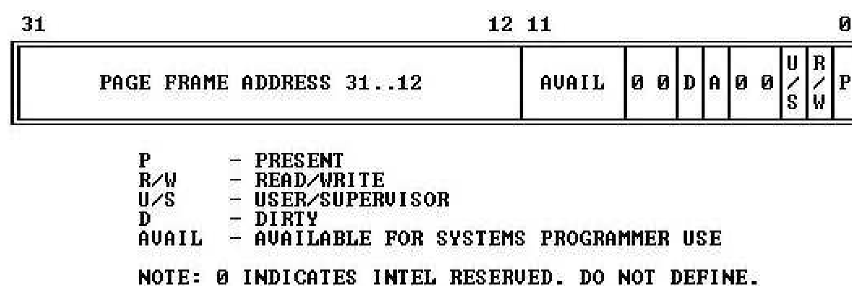


Figure 4.1: Page Table Entry of the i386 in the Linux Kernel

A major unsolved problem with the conventional incremental checkpointing is the efficiency of the checkpoint interval. In periodic interval checkpointing, the checkpoint cost is constant for a checkpoint interval irrespective of the amount of information to be stored. But in case of incremental checkpointing, cost of checkpointing is depends on the amount of information that has modified or the number of pageas that has modified.

In next section, we are going to determine a dynamic interval for checkpointing based on the expected execution time.

4.1 BASICS OF PROBABILITY

Here, we are deriving the expected execution time by using the theory of probability and calculus. Some definitions of probability:

-*Conditional probability* is the probability of some event A, given the occurrence of some other event B. Conditional probability is written P(A|B), and is read "the probability of A, given B".

-*law of total expectation* states that if X is an integrable random variable (i.e., a random variable satisfying E(| X |) $< \infty$) and Y is any random variable, not necessarily integrable, on the same probability space, then E(X) = E(E(X|Y)).
i.e., the expected value of the conditional expected value of X given Y is the same as the expected value of X.

The conditional expected value E(X | Y) is a random variable in its own right, whose value depends on the value of Y. Notice that the conditional expected value of X given the event Y = y is a function of y. If we write $E(X|Y = y) = g(y)$ then the random variable $E(X|Y)$ is just $g(Y)$.

If X is an event that is running on a system subjected to the time t, then the total expected time to finish that event is given [34, 9, 19] by

$$E(t) = \int_0^\infty X(t)\,dt$$

4.2 NOTATIONS AND ASSUMPTIONS

The following notations are used in the derivation of total expected execution time as shown in Table 4.1 .

t	total work requirement time of a process
t_i	work requirement time of interval i
r	recovery cost of a process
c_i	checkpointing cost of interval i
λ	failure rate
m	number of the modified pages of a process
c_p	checkpointing cost of a page
$f(k)$	probability dencity function of a failure at elapsed time k
$T(t)$	expected execution time of a process
$T_c(t,c)$	expected execution time of a process with page-level incremental checkpointing
$Rec_{skip}(t)$	expected recovery time of a process without checkpointing
$Rec_{take}(t)$	expected recovery time of a process with page-level incremental checkpointing

Table 4.1: Notations used in algorithm

The following is the system model used for our experiment on the expected execution time with and without checkpointing. First and foremost, the expected total execution time of a process can be defined as the processing time from the beginning of its execution to its completion.

Let $T(t)$ denote the expected execution time of a process and t be the "work requirement " of the process. Note that in the absence of any failures [34, 9, 19],
$T(t)$ = t.

We assume that the expected total execution time of a process with incremental checkpointing, $T_c(t,c)$, consists of the execution time of n intervals and time for incremental checkpointing is at the end of each interval.

A checkpoint interval is the duration between two checkpoints. That is, it begins when a checkpoint is established and ends when the next checkpoint is established.

Let $T_c(t_i, c_i)$ denote the expected time required to execute the interval i and take an incremental checkpoint. Obviously,

$$\sum_{i=1}^{n} T_c(t_i, c_i) = T_c(t, c)$$

in the absence of failures. Note that $T_c(t_i, c_i)$ is a random variable, even though t_i is fixed and the interval i could be executed in time $T_c(t_i, c_i)$ as long as, there no failure occurs.

Here, we assumed that failures can occur during normal execution as well as during checkpointing. We also assumed that failures occurs according to a Poisson process at rate, λ. These are commonly accepted assumptions, particularly when failures are known to occur as a result of many different reasons. Further, we assume that failures are detected as soon as they occur.

4.3 EXPECTED EXECUTION TIME WITHOUT CHECKPOINTING

Clearly, the execution time t of a process is identical to $T(t)$ in the absence of failures without checkpointing. However, in the presence of failure, if a failure occurs before the completion of a process, then a recovery cost r is incurred to resume its execution from its beginning of the process. Fig 4.2 shows an example of a process which is being executed without checkpointing and where a failure occurs.

In Fig 4.2, if a failure occurs before the end of execution (k < t) , the expected execution time of a process without checkpointing $T(t)$, can be calculated by the disjunct components, k, r and $T(t)$. In this case, the process must restart from the beginning, which means that the remaining work requirement is t and its expected execution time is $T(t)$.

We assumed that the failures occur according to a Poisson process at rate λ, thus $F(K) = 1 - e^{-\lambda k}$ and the $f(k) = \lambda e^{-\lambda k}$, then the expected execution time of a process without checkpointing is given as Theorem 4.1.

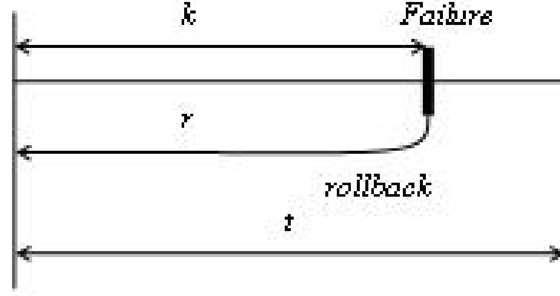


Figure 4.2: Execution without checkpointing

Theorem 4.1: *The expected execution time $T(t)$ of a process without checkpointing is given by*

$$T(t) = \frac{\left(e^{\lambda t} - 1\right)(1 + \lambda r)}{\lambda} \tag{4.1}$$

PROOF: Formally, the conditional expected execution time is written as

$$T(t) = \begin{cases} t & if \quad k \geq t \\ k + r + T(t) & otherwise \end{cases}$$

By the law of total expectation,

$$T(t) = \int_t^{\infty} tf(k)\,dk + \int_0^t (k + r + T(t))f(k)\,dk$$

Solving the above equation we obtain,

$$T(t) = \frac{\int_0^{\infty} tf(k)\,dk + \int_0^t (k + r - t)f(k)\,dk}{1 - \int_0^t f(k)\,dk}$$

Since $\int_0^{\infty} tf(k)\,dk =$ t, rearranging with respect to $T(t)$, we obtain,

$$T(t) = \frac{t + \int_0^t (k + r - t)f(k)\,dk}{1 - \int_0^t f(k)\,dk}$$

Finally the probabilistic density function of failure f(k) is $\lambda e^{-\lambda k}$, we will get,

$$T(t) = \frac{\left(e^{\lambda t} - 1\right)(1 + \lambda r)}{\lambda}$$

4.4 EXPECTED EXECUTION TIME WITH PAGE LEVEL INCREMENTAL CHECKPOINTING

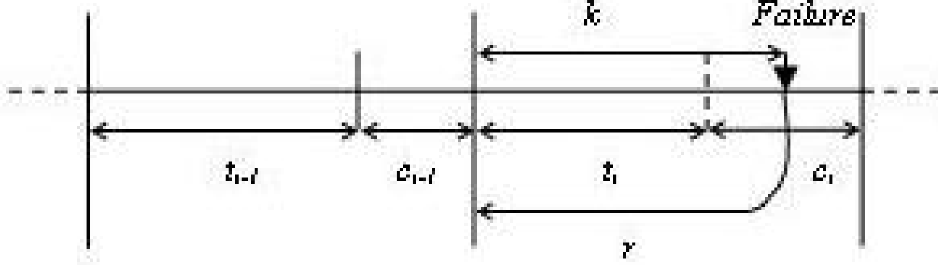


Figure 4.3: Execution with Incremental Checkpointing

Fig 4.3 shows the execution with pagelevel incremental checkpointing. When a failure occurs during a checkpoint interval, the process must roll back and reprocess its execution from the beginning of the interval. If no failure occurs during normal execution and checkpointing, then the expected execution time of the interval i is identical to $(t_i + c_i)$. However, if a failure occurs during a checkpoint interval i, recovery time r to roll back to the beginning of the interval is required. Therefore, if a failure occurs before the end of interval i, $k < (t_i + c_i)$, the expected execution time of an interval with a pagelevel incremental checkpoint, $T_c(t_i, c_i)$ is given by Theorem 4.1.

Theorem 4.2: . If we assume the failures occur according to the Poisson process with rate λ and that failures can occur during checkpointing, the expected execution time $T_c(t_i, c_i)$ of the interval i with a pagelevel incremental checkpoint is given by

$$T_c(t_i, c_i) = \frac{\left(e^{\lambda(t_i+c_i)} - 1\right)(1 + \lambda r)}{\lambda} \tag{4.2}$$

where $c_i = mc_p$

PROOF: . By substituting $(t_i + c_i)$ for t in Eq.(4.l), we obtain Eq.(4.2).

When the expected total execution time $T_i(t_i, c_i)$ is divided into n intervals and an incremental checkpoint is taken at the end of each interval, the expected total execution time with pagelevel incremental checkpoints can be expressed as follows.

$$T_c(t,c) = \sum_{i=1}^{n} \frac{\left(e^{\lambda(t_i+c_i)} - 1\right)(1+\lambda r)}{\lambda}$$

If we assume that the checkpoints are equally spaced, and the checkpointing cost is constant c, then the above equation becomes much easier. However, these assumptions do not reflect the actual characteristics of incremental checkpointing. In incremental checkpointing, the checkpointing cost c_i is not a constant value, and furthermore, the checkpointing interval cannot be a constant duration. For instance, c_i can vary under the modification pattern of a process memory pages that we dont know. Thus, this equation is not an appropriate way of deriving an efficient checkpointing interval.

4.5 DYNAMIC INTERVAL DETERMINATION

In this section, we derive an efficient interval of pagelevel incremental checkpointing by the cost analysis of expected recovery time. We will analyze the expected recovery time and present the dynamic interval determination mechanism on pagelevel incremental checkpointing.

4.5.1 Cost Analysis of Expected Recovery Time

Fig 4.4 shows the example situation of a process with pagelevel incremental checkpointing, c_{i-1} is a checkpointing time of the $(i-1)^{th}$ interval, the t_i is a processing time of the ith interval, and the c_i is an estimated checkpointing time of current execution point of the process. In this situation, we can take or skip an incremental checkpoint at this time.

When a process takes an incremental checkpoint at this time, the process waits for incremental checkpointing time c_i. The process may extend the execution time of the process because of c_i when no failure exist, but if failure does occur, the recovery time may be reduced by the current checkpoint. We derive the expected recovery time of a process in cases where we have to decide whether to skip or take a checkpoint.

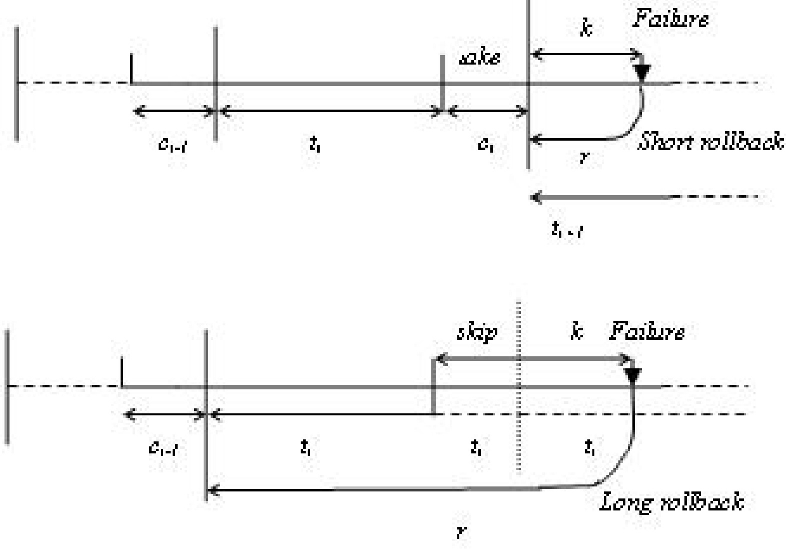


Figure 4.4: Example Situation with Incremental Checkpoinitng

Skip: The expected recovery time from the figure and the law of total expectation for checkpoint skip is given by ,

$$Rec_{skip}(t_i, c_i) = \int_0^{\infty} (k + r + T(t))f(k)\, dk$$

Take: The expected recovery time for checkpoint take is derived as follows ,
From the Fig 4.4 and law of conditional expectation,

$$Rec_{take} = \begin{cases} k + r + T(t_i + c_i) & if \quad k < c_i \\ k + r & otherwise \end{cases}$$

From the law of total expectation ,

$$Rec_{take}(t_i, c_i) = \int_0^{c_i} (k + r + T(t))f(k)\, dk + \int_{c_i}^{\infty} (k + r)f(k)\, dk$$

Finally, $Rec_{take}(t_i, c_i) - Rec_{skip}(t_i, c_i)$ represents a decider $Dec(t_i, c_i)$ of the two alternatives, we obtain,

$$Dec(t_i, c_i) = (1 - e^{-\lambda c_i})T(t_i + c_i) - T(t_i)$$

By calculating $Dec(t_i, c_i)$, we can decide whether to skip or take. If $Dec(t_i, c_i)$ is positive, $Rec_{take}(t_i, c_i)$ is larger than $Rec_{skip}(t_i, c_i)$, then we may skip an incremental checkpoint. Otherwise, we may take an incremental checkpoint. For example, if t_i is nearly zero, then $Dec(0, c_i)$ converges to positive because $e^{-\lambda c_i} < 1$ and $T(c_i) > 0$, so we skip a checkpoint. For another example, if t_i is nearly ∞, then $Dec(\infty, c_i)$ converges to negative because $e^{-\lambda c_i} > 0$ and $T(t_i) > 0$, so we take a checkpoint.

4.5.2 Interval Determination Mechanism

Since $Dec(t_i, c_i)$ changes over t_i and c_i, we need to calculate $Dec(t_i, c_i)$ when t_i or c_i is changed. Even c_i is not changing on run-time, we still need to calculate the $Dec(t_i, c_i)$. The checkpointing cost c_i is increasing when some pages are modified. Therefore, we need to find an appropriate x to make $Dec(t_i + x, c_i)$ to be zero.

$$x = \frac{1}{\lambda} ln \left(\frac{e^{-\lambda c_i}}{2 - e^{\lambda c_i}} \right) - t_i$$

Here, x represents the next checkpointing time and thus can be used to determine the checkpointing interval. For efficient calculation of the above equations, we attached the interval determination mechanism to the page-fault handler. Algorithm 4.1 shows the mechanism in the page-fault handler and timer expire routine.

Algorithm 4.1:

Interval determination mechanism

-$In\ page\ fault\ handler$

if a page has modified **then**

  Increase m

  Compute Dec

  if $dec < 0$ **then**

    Take an incremental checkpoint

  else

    Compute new x to take incremental checkpoint

    Set the timer to new x value

  end If

end If

-*In Timer Routine*

If The timer is x **then**

Take an incremental checkpoint.

Compute new x

Set the timer to new x value

end If

In Algorithm 4.1, m denotes the number of modified pages on the checkpoint interval and x is the adequate time to make the $Dec(t_i+x, c_i)$ to zero. The timer is set to the x to take an incremental checkpoint. By doing so, we can determine the efficient checkpoint interval on pagelevel incremental checkpointing.

4.6 PERFORMANCE EVALUATION

Here, we are going to discuss the performance of the *dynamic interval determination for the pagelevel incremental checkpointing* facility based on the efficient interval determination mechanism. We applied this algorithm, to Matrix Multiplication and Quick Sort, to compare the execution times of periodic interval checkpointing and dynamic interval checkpointing. To compare the checkpoint overhead, we measured the average execution time of a process for these applications.

The average execution time of each application with periodic pagelevel incremental checkpointing and the dynamic pagelevel incremental checkpointing with efficient interval determination is shown in Fig(4.5, 4.6). $Dec(t_i, c_i)$ is the calculated discriminant based on the expected recovery time of a process decision when skip or take an incremental checkpointing. Thus, it means the efficiency of checkpoint placement. In periodic checkpointing the process should take the checkpoint, though it is not needed. Where as in dynamic interval determinism, it depends on the *decide* value which is calculated based on *expected recovery time* and it avoids unnecessary checkpoints. Obviously, it reduces the checkpoint overhead and execution time of the process.

In Fig 4.5, the proposed algorithm is applied on Matrix Multiplication and is compared with periodic incremental checkpointing algorithm and same with Quick Sort in Fig 4.6.

In this result, the checkpoint placement of the dynamic interval pagelevel incremental

checkpointing is more efficient than any other periodic pagelevel incremental checkpointing.

These results shows that the execution time with dynamic interval pagelevel incremental checkpointing had been significantly reduced compared with periodic pagelevel incremental checkpointing. It is noted that the dynamic interval pagelevel incremental checkpointing with interval determination reduce the execution time of a process by about 20% more than using periodic pagelevel incremental checkpointing.

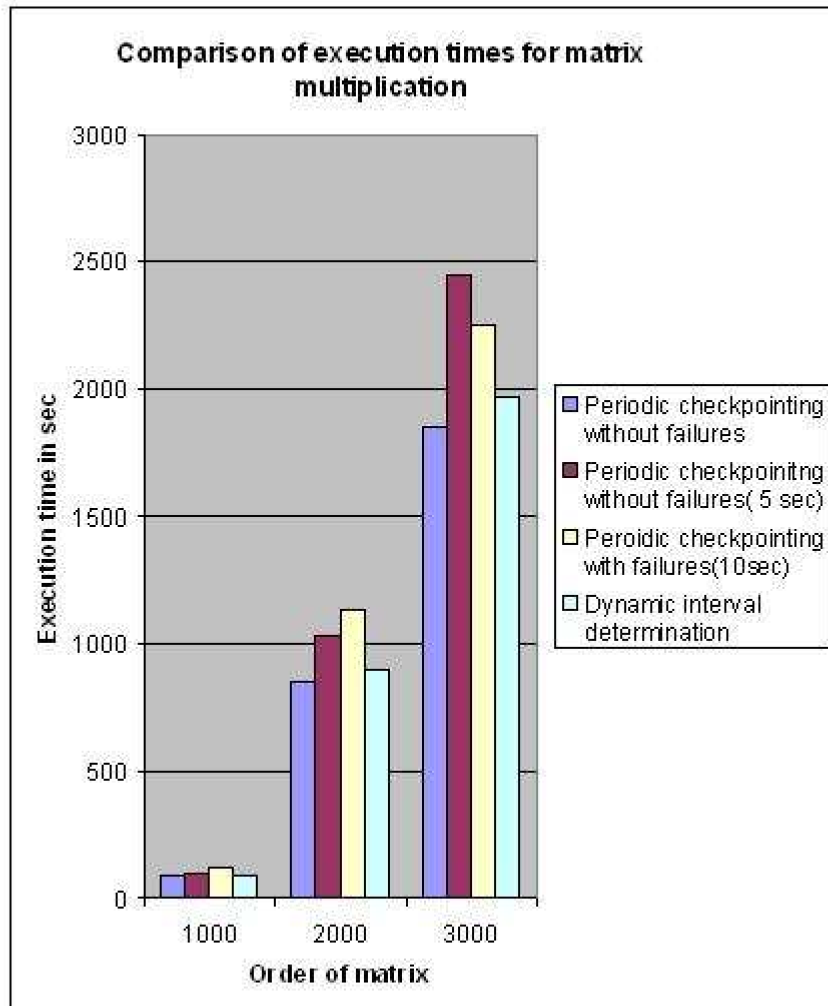


Figure 4.5: Comparison of average execution times for matrix multipication

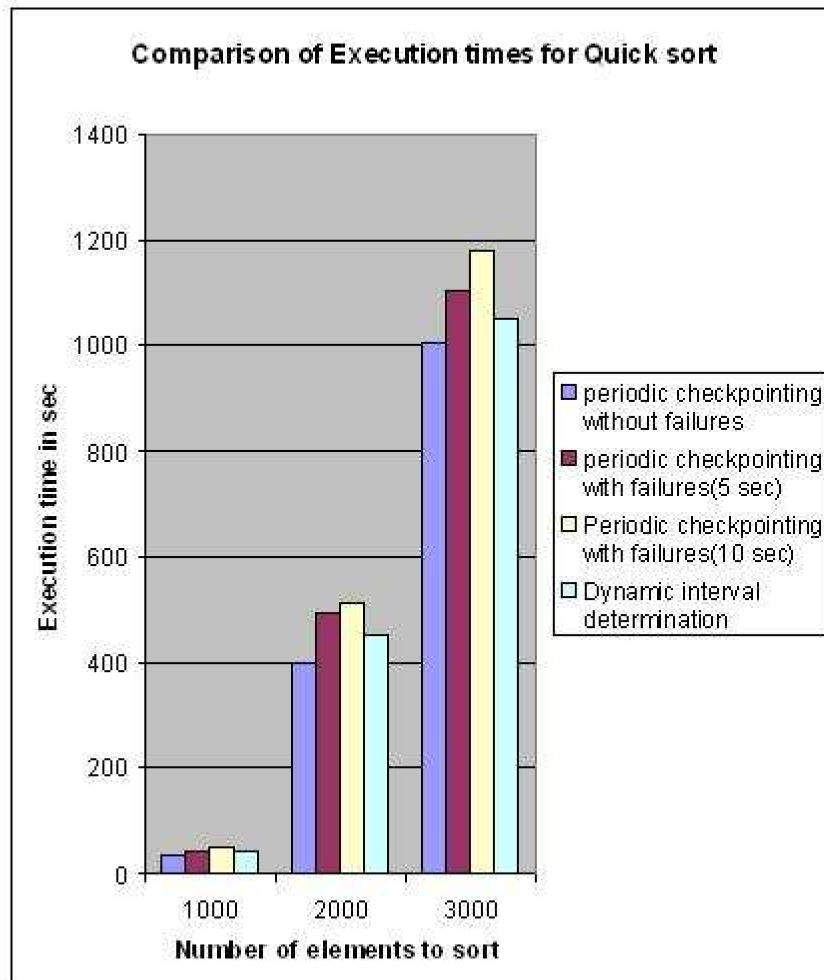


Figure 4.6: Comparison of average execution times for quick sort

Chapter 5

CONCLUSION AND FUTURE WORK

This thesis presents two algorithms on checkpointing. One is coordinated checkpoint algorithm for reducing the number messages per process that are needed for checkpoint coordination and there by reducing netwotk congestion. Another one is "Dynamic interval determination for pagelevel incremental checkpointing", which reduces the checkpointing overhead by eliminating unnecessary checkpoints based on expected recovery time. Finally, section 5.2 discuss the scope for future work.

5.1 THESIS SUMMARY

The following are the main contributions of the thesis.

5.1.1 A checkpointing algorithm for reducing coordination overhead

Checkpointing is a fault tolerant technique, which allows the system to run its applications in less time in the event of failures also. But checkpointing adds the overhead to the Distributed system. It adds the overheads like checkpointing overhead and context saving overhead. System is getting coodination overhead, mainly by the messages that are needed for checkpoint ordination. Existing algorithm is taking three messages per process(request,reply and final decision messages) for coordinating a single checkpoint. If the more number of process are in the system then more number of coordination messages will be there. Obviously, these unnessary messages increases the network traffic. This is not advisable. We should reduce the network traffic by reducing the number of coordination messages. The proposed algorithm is reducing the number of coordination messages by eliminating unnecessary reply messages. On succesful checkpoint proposed algorithm is taking only two messages per process for checkpoint coordination. In this we reduced number of messages that are needed for checkpoint coordination.

5.1.2 Dynamic interval determination for pagelevel incremental checkpointing

Checkpointing is also caused to increase the execution time of the process by adding its checkpoint context saving overhead. Existing periodic checkpointing algorithms are taking checkpoints, though it is not needed all the time. These unnecessary checkpoints

increasing the execution time of the process. To overcome this, increnetal checkpointing has proposed, which saves only the pages that are modified from previous checkpoint. We have to mtake the checkpoints in such a way that overhead and recovery cost should be minimum. Our proposed algorithm is achieving this by determining interval dynamically for taking checkpoints based on the expected recovery time. This expected recovery is calculated by using the principles of probability and calculus.

For calculating *decide* value, we are using expected recovery time by skipping checkpoint and by taking checkpoint. After getting this *decide* value, the algorithm dtermines whether to skip the checkpoint or to take the checkpoint. By eliminating some checkpoints the execution time of the program will reduce. Proposed algorithm takes checkpoints only when taking the checkpoint is reasonable.

5.2 FUTURE WORK

5.2.1 Coordinated checkpoint algorithm

The coordinated checkpoint algorithm that we were proposed is reducing the number of messages that are needed for checkpoint coordination. In the proposed algorithm, the processes that are involved in taking checkpointing will stop their executions until the finishing of checkpoint process. Another thing in the proposed algorithm is some processes are forced to take the checkpoints.

This halting problem can be overcome by creating the *child process* for each process. The job of the child processes is saving parent process execution to the stable storage when the execution of the parent process is started. This will reduce the execution time of the process by performing the context saving in parallel with the execution.

5.2.2 Dynamic interval determination for pagelevel incremental checkpointing

The algorithm "Dynamic interval determination for pagelevel incremental checkpointing" minimizes the execution time of a process in compared with any other periodic checkpointing. The algorithm that we were proposed can be extended to the *Distributed System*. For extending it to distributed system one should deal with all designing issues of distributed system and issues that are related to checkpointing for distributed system.

Bibliography

[1] ACHARYA A. and BADRINATH B. R., "Recording distributed snapshots based on causal order of message delivery," *Information Processing Letters 44*, pp. 317–321, 1992.

[2] ALVISI and L., "Understanding the message logging paradigm for masking process crashes," Ph.D. dissertation, Cornell University, January 1998.

[3] BORG A., BAUMBACH, J., and GLAZER S., "A message system supporting fault tolerance." In Proc. of the Symp. on Operating Systems Principles (ACM SIGOPS), Oct 1983, pp. 90–99.

[4] BORG A., BLAU W., HERRMANN F., and OBERLE W., "Fault tolerance under unix," *ACM Computing Surveys*, vol. 7, pp. 1–24, Oct 1989.

[5] CAO G. and SINGHAL M., "On coordinated checkpointing in distributed system." *IEEE Trans, on Parallel and Distributed Systems 9, 12*, pp. 1213–1225, December 1998.

[6] CAO G and SINGHAL M., "Mutable checkpoints: A new checkpointing approach for mobile computing systems." *IEEE Trans, on Parallel and Distributed Systems 12, 2*, pp. 157–172, February 2001.

[7] CHANDY K. M. and LAMPORT, "Distributed snapshots: Determining global states in distributed systems," *ACM Trans on Computer Systems*, vol. 3, no. 1, pp. 63–75, February 1985.

[8] CRITCHLOW C and TAYLOR K., "The inhibition spectrum and the achievement of causal consistency," Cornell University," Tech. Rep. TR 90-1101, February 1990.

[9] DANIEL NURMI, RICH WOLSKI, and JOHN BREVIK, “Modelbased checkpoint scheduling for volatile resource environments,” UCSB Computer Science, Technical Report 2004-25, 2004.

[10] ELNOZAHY and E. N. MaNetHo, “Fault tolerance in distributed systems using rollback-recovery and process replication,” Ph.D. dissertation, Rice University, Texas, Austin, October 1993.

[11] ELNOZAHY E. N., ALVISI L., WANG Y. M., and JOHNSON D. B, “A survey of rollback-recovery protocols in message-passing systems,” *ACM Computing Surveys 34, 3*, pp. 375–408, September 2002.

[12] ELNOZAHY E. N., JOHNSON D. B., and ZWAENEPOEL W. , “The performance of consistent checkpointing.” In Proc. of IEEE Symp. on Reliable Distributed Systems, October 1992, pp. 39–47.

[13] HELARY J. M., MOSTEFAOUI A., NETZER R. H., and RAYNAL M., “Preventing useless checkpoints in distributed computations.” In Proc. of the 16th Symposium on Reliable Distributed Systems, 1997, pp. 183–190.

[14] K. Ll, J.F.NAUGHTON, and J.S.PLANK, “low latency, concurrent checkpointing for parallel programs,” *IEEE Tran. on parallel and distributed systems, 5,8*, pp. 874–879, August 1994.

[15] S. KALAISELVI and V. RAJARAMAN, “A survey of checkpointing algorithms for parallel and distributed computers,” *Sadhana*, vol. 25, no. 5, pp. 489–510., october 2000.

[16] KOO R. and TOUEG S., “Checkpointing and rollback-recovery for distributed systems,” *IEEE Trans, on Software Engg. 13, 1*, pp. 23–31, 1987.

[17] LAI T. H. and YANG T. H., “On distributed snapshots,” in *Information Processing Letters 25*, 1987, pp. 153–158.

[18] LAMPORT L. and M. P. M., “Synchronizing clocks in the presence of faults.” *Journal of the ACM 32*, pp. 52–78, January 1985.

[19] LEV FINKELSTEIN, SHAUL , and EHUD RIVLIN, "Optimal schedules for parallelizing anytime algorithms: The case of shared resources," *Journal of Artificial Intelligence Research 19*, pp. 73–138, August 2003.

[20] M. SINGHAL and N.G. SHIVARATRI, *Advanced concepts in Operating Systems.* New Delhi: TATA McGRAW-HILL, 2005.

[21] MANIVANNAN D. and SINGHAL M., "A low-overhead recovery technique using quasi-synchronous checkpointing," in *In Proc. of Distributed Computing Systems.* ACM pub, May 1996, pp. 100–107.

[22] PLANK J., Li K., and PUENING M. A., "Diskless checkpointing," *IEEE Trans, on Software Engg. 9, 10*, pp. 972–986, October 1998.

[23] PLANK J. S., "Efficient checkpointing on MIMD architechture," Ph.D. dissertation, Princeton University, 1993.

[24] PLANK J. S., Xu J., and NETZER R. H., "Compressed differences: An algorithm for fast incremental checkpointing," University of Tennessee at Knoxville," Tech. Rep. CS-95-302, 1995.

[25] PRAKASH R. and SINGHAL M., "Low-cost checkpointing and failure recovery in mobile computing systems," *IEEE Trans, on Parallel and Distributed Systems 7, 10*, pp. 1035–1048, October 1996.

[26] RANDELL B. , "System structure of software fault tolerance," *IEEE Trans, on Software Engg. 1, 2*, pp. 220–23, 1975.

[27] SILVA L. M. and SILVA J. G., "The performance of coordinated and independent checkpointing." In Proc. of Intl. Parallel and Distributed Processing Symposium, 1999, pp. 88–94.

[28] SISTLA A. P. and WELCH J. L., "Efficient distributed recovery using message logging," *IEEE/ACM Trans, on Networking 4, 5*, pp. 785–795, 1996.

[29] STROM R. E. and YEMINI S., "Optimistic recovery in distributed systems," *ACM Trans. on Computer Systems 3, 3*, pp. 204–226, August 1985.

[30] T. A. WELCH, "A technique for high performance data compression," *IEEE Computer,17*, pp. 8–19, June 1984.

[31] TAMIR Y. and SEQUIN C. H., "Error recovery in multicomputers using global checkpoints." Intl. Conf. on Parallel Processing, 1984, pp. 32–41.

[32] WANG and Y. M., "Consistent global checkpoints that contain a given set of local checkpoints," *IEEE Trans, on Computers 46,4*, pp. 456–468, April 1997.

[33] WANG, Y. M., and FUCHS W. K., "Optimistic message logging for independent checkpointing in message passing systems." In Proc. of IEEE Symp. on Reliable Distributed Systems, October 1992, pp. 147–154.

[34] YIBEI LING, JIE MI, and XIAOLA LIN, "A variational calculus approach to optimal checkpoint placement," *IEEE Transactions on computer*, vol. 50, no. 7, July 2001.