Reliability in High Performance Distributed Computing Systems

Adnan Agbaria
Reliability in High Performance Distributed Computing Systems

Research Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Adnan Agbaria

Submitted to the Senate of the Technion — Israel Institute of Technology
Teshri, 5763 Haifa September, 2002
The research thesis was done under the supervision of Dr. Roy Friedman in the department of Computer Science

I wish to express my deepest gratitude to Dr. Roy Friedman for his invaluable help and his professional guidance throughout this research. Prof. Hagit Attiya, Prof. Jim Plank, Roman Vitenberg, Raja Said, and Ari Freund have collaborated with us on parts of this research. Moreover, I would like to thank Alon Kama for his grammar comments. I wish to thank the people of the faculty of Computer Science, with whom I have worked together for a long time, and who made my years in the Technion an enjoyable experience.

Lastly, I would like to thank my wife Soona and my daughter Maram for their patients and encourages during the research.

THE GENEROUS FINANCIAL HELP OF THE TECHNION, THE MINISTRY OF SCIENCE, VATAT, BAR-NIR BERGREEN SOFTWARE TECHNOLOGY CENTER OF EXCELLENCE, AND GUTWIRTH FUND IS GRATEFULLY ACKNOWLEDGED
## Contents

List of Symbols and Abbreviations 2

1 Introduction 3

2 Related Work 8

2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2 Comparing with Existing Systems 8

2.3 Heterogeneous Checkpointing . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.3.1 C/R in Java and Java Mobility 11

2.4 Distributed Checkpointing 12

2.5 Performance Analysis 14

2.6 Fault-Tolerance by Replication 14

3 The Basic Framework 16

3.1 The System Model 16

3.1.1 Practical Considerations 17

3.1.2 Distributed Checkpointing 18

4 The Starfish System 19

4.1 Overview 19

4.2 Starfish Architecture 20

4.2.1 Daemon Internals 21

4.2.2 Application Process Internals 22

4.2.3 Interaction Between Daemon and Application Process 24

4.3 Starfish Features 24

4.3.1 Starfish Manageability, Dynamicity, and High Availability 24

4.3.2 Applications Dynamicity and Fault Tolerance 26

4.4 The Starfish Control Center 27

4.4.1 The Menu Commands 27

4.5 Experience Work 30

4.5.1 Checkpoint time 30

4.5.2 Round-Trip Delay 31

4.5.3 System Overheads 31

4.6 Using Starfish with NetSolve [17] 32

4.6.1 Putting it all Together 33

4.6.2 Benefits of This Architecture 33

4.6.3 Performance Case Studies 34

5 Heterogeneous Checkpointing 37

5.1 Overview 37

5.2 The OCaml Virtual Machine 37
## Contents (Cont.)

5.2.1 Main Memory Areas ........................................ 38  
5.2.2 Data Representation ........................................ 39  
5.2.3 Threads ....................................................... 39  
5.2.4 Garbage Collection .......................................... 40  
5.3 Heterogeneous Checkpoint and Restart .......................... 40  
5.3.1 Checkpoint Issues ........................................... 41  
5.3.2 The Checkpoint Mechanism .................................. 42  
5.3.3 Restart Issues ................................................ 44  
5.3.4 The Restart Mechanism ....................................... 45  
5.4 Evaluating Heterogenous C/R .................................... 45  
5.4.1 Advantages, Disadvantages, and Restrictions ............... 45  
5.4.2 Experimental Results ........................................ 47  
5.5 Heterogeneous C/R in Starfish ................................ 50  

6 Distributed Checkpointing ......................................... 51  
6.1 Overview ......................................................... 51  
6.2 Definitions and Notations ....................................... 51  
6.3 Classifying Executions by Rollback Distance ................... 55  
6.4 A d-BC Checkpointing Protocol ................................. 62  
6.4.1 Data Structures .............................................. 62  
6.4.2 The Protocol .................................................. 63  
6.4.3 Computing an Overlapping Class ............................. 66  
6.4.4 Protocol Properties .......................................... 66  

7 Evaluating Distributed Checkpointing Protocols .................. 70  
7.1 Overview ......................................................... 70  
7.2 Definitions and Notations ....................................... 70  
7.3 The Overhead Ratio .............................................. 72  
7.3.1 Computing $\Gamma_k$ Using Markov Chains .................... 73  
7.3.2 Computing the Overhead Ratio ................................ 78  
7.3.3 Numerical Results ............................................ 80  
7.4 Performance Analysis of Checkpointing Protocols ............... 82  
7.4.1 Distributed Checkpointing Protocols ....................... 83  
7.4.2 Comparing Distributed Checkpointing Protocols .......... 85  

8 Summary and Future Work ......................................... 89  
8.1 Review .......................................................... 89  
8.2 Future Work ..................................................... 91  
8.2.1 Starfish ....................................................... 91  
8.2.2 Heterogeneous Checkpointing ................................ 91  
8.2.3 Distributed Checkpointing Protocols ....................... 92  
8.2.4 Performance Analysis ......................................... 92
## Contents (Cont.)

A  Local Checkpointing in Starfish  
   A.1  Checkpointing ................................................. 102  
   A.2  Restarting ...................................................... 102  

B  Implementing $d$-BC in Starfish  
   B.1  Data Structures .............................................. 105  
   B.2  The Protocol .................................................... 106  
      B.2.1  Find Class Borders .................................... 106  

C  A Matlab Program for Computing the Overhead Ratio  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  Local Checkpointing in Starfish</td>
<td>102</td>
</tr>
<tr>
<td>A.1  Checkpointing</td>
<td>102</td>
</tr>
<tr>
<td>A.2  Restarting</td>
<td>102</td>
</tr>
<tr>
<td>B  Implementing $d$-BC in Starfish</td>
<td>105</td>
</tr>
<tr>
<td>B.1  Data Structures</td>
<td>105</td>
</tr>
<tr>
<td>B.2  The Protocol</td>
<td>106</td>
</tr>
<tr>
<td>B.2.1  Find Class Borders</td>
<td>106</td>
</tr>
<tr>
<td>C  A Matlab Program for Computing the Overhead Ratio</td>
<td>110</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Periodic checkpoints for fault-tolerance. ................................................................. 4  
2.1 Examples of primary/backup and active replication approaches. ................................. 15  
3.1 A process of a distributed system. ............................................................................. 17  
4.1 Low level view of Starfish. .......................................................................................... 20  
4.2 Starfish architecture. ................................................................................................. 21  
4.3 An abstract view of Starfish. ...................................................................................... 22  
4.4 The login dialog box of SCC. ..................................................................................... 27  
4.5 The ”Host” view of SCC. .......................................................................................... 28  
4.6 The submit dialog box of SCC. .................................................................................. 29  
4.7 Checkpointing in Starfish. ......................................................................................... 30  
4.8 Round-trip delay vs. data size. ................................................................................... 31  
4.9 Layers overhead for sending and receiving messages. .................................................. 32  
4.10 The structure of NetSolve applications. ................................................................... 32  
4.11 NetSolve with checkpointing. ................................................................................... 33  
4.12 Performance of \texttt{dmatmul} on the CLUSTER environment. ............................... 35  
4.13 Performance of \texttt{dmatmul} on the LOCAL and WIDE environments. .................. 36  
5.1 A blocked type with its header as defined by OCaml. ................................................... 39  
5.2 The C/R module is located inside the OCaml virtual machine. ................................. 41  
5.3 Relevant source code of the checkpoint function in OCVM. ....................................... 43  
5.4 Relevant source code of the restart function in OCVM. ............................................. 46  
5.5 Comparing the running time of matrix multiplication and FFT respectively with and without checkpointing on \texttt{rodrigo}. ............................................................. 48  
5.6 Comparing the restart time on different platforms. ..................................................... 48  
5.7 Timing the substantial parts on \texttt{rodrigo} vs. various checkpointed data size. .......... 49  
5.8 Comparing local homogeneous and heterogeneous checkpointing in Starfish. ............ 50  
6.1 An execution example: $\xi_1 = (m_4,m_5)$ is a Z-path from $C_{1,1}$ to $C_{3,2}$. $\xi_2 = (m_7,m_4)$ is a Z-cycle from $C_{2,2}$ to itself, where $P_1,P_2 \in \xi_2$. ................................................................. 52  
6.2 Execution example with two cycle classes. ................................................................ 54  
6.3 An execution example for illustrating the proof of Theorem 6.3.1. .......................... 57  
6.4 Relationship between the various classes. .................................................................. 57  
6.5 The longest rollback for $d$-BC. ................................................................................ 58  
6.6 An execution example for illustrating the proof of Theorem 6.3.3. ........................... 61  
6.7 Process $p$ tries to advance $R_p$ to include checkpoint $C_{q,j}$. ............................... 64  
6.8 Process $p$ advances $R_p$ to include checkpoint $C_{q,j}$. .......................................... 65  
6.9 The cycle graph for the execution of Figure 6.2. This graph contains two strongly connected components. ................................................................. 66
6.10 An execution example with a partition \( I = \{I_1, I_2, I_3, I_4\} \) of the checkpoint intervals between \( R_p \) and \( Q_p \). .................................................. 69

7.1 Specifying the parameters for the events in process \( p \). .......................... 71
7.2 A possible execution of \( I_{p,i} \). ................................................................. 73
7.3 A Markov Chain represents \( I_{p,i+1} \) execution in the 0-rollback class. .... 74
7.4 A Markov Chain represents \( I_{p,i+1} \) execution in 1-rollback class. ........ 75
7.5 A Markov Chain represents \( I_{p,i+1} \) execution in 2-rollback class. ......... 76
7.6 A Markov Chain represents \( I_{p,i+1} \) execution in 3-rollback class. ......... 77
7.7 A Markov Chain represents \( I_{p,i+1} \) execution in \( k \)-rollback class. ....... 78
7.8 SMALL and BIG environments, for every \( v(k) \) there is an optimum checkpoint interval \( (T) \) – note difference in scale. .......................................................... 81
7.9 The overhead ratio VS. \( T \) in SMALL. ....................................................... 82
7.10 The overhead ratio is affected by other parameters like \( F \) and \( n \). .......... 82
7.11 The overhead ratio increases linearly with \( d \) in \( d \)-BC executions. ....... 86
7.12 Comparing the SaS and C-L protocols with various number of processes. .. 86
7.13 Comparing checkpointing protocols of different classes with equal probabilities for rollback. .......................................................... 87
7.14 Comparing checkpointing protocols of different classes with decreasing prob-
abilities for rollback. .......................................................... 88

A.1 A memory areas of a running process in Linux OS .................................. 103
A.2 Save the data segment in the checkpoint file. ......................................... 104
A.3 Save the stack segment in the checkpoint file. ....................................... 104
A.4 Restore the stack segment from a checkpoint file. .................................. 104

B.1 Process \( p \) receives a data message from another process. .................... 106
B.2 The code of \texttt{findClassBorders} as implemented in Starfish. ............... 107
List of Tables

4.1 Summary of message types in Starfish ........................................... 23
5.1 A list of different machines that have been tested with heterogeneous C/R . 47
Abstract

The increase in microcomputing power and network speed offer the possibility of replacing a supercomputer with a cluster of small computers, thereby reducing the cost of high-performance computing. However, unlike traditional supercomputers, a distributed cluster is prone to partial hardware failures, and the probability of such a failure grows linearly with the number of nodes. Therefore, a cluster of workstations for high performance computing cannot replace a supercomputer functionality without supporting reliability.

This thesis addresses practical and theoretical approaches for providing reliability in high performance distributed computing systems. We start by presenting a framework for defining the distributed computing model that we consider. We define the model with checkpoint and recovery (changeably, restart) events, and then conditions on which a set of checkpoints is consistent. Checkpointing an application is the act of saving its state during execution on stable storage so that if the application fails, it can be recovered from the last saved state.

Following this, we present the Starfish system, an environment for executing dynamic (and static) MPI programs on a cluster of workstations. Starfish is unique in being efficient, fault-tolerant, highly available, and dynamic as a system internally, and in supporting fault-tolerance and dynamicity for its application programs as well. Starfish achieves these goals by combining group communication technology with checkpoint/restart, and uses a novel architecture that is both flexible and portable and keeps group communication outside the critical data path, for maximum performance.

We also present a mechanism that enables Starfish to be a heterogeneous reliable distributed system. This is the heterogeneous checkpoint/restart module that allows to recover an application from a saved state that was taken in a hardware architecture and/or operating system that can be different from those in the machine on which it is recovered.

Next, we present a classification of distributed executions based on rollback propagation during recovery. Specifically, an execution is \( k \)-rollback, if \( k \) indicates the maximum number of checkpoints that need to be rolled back for recovery. We show a new class of execution, called \( d \)-bounded cycles, and show that this execution is \( (n - 1)d \)-rollback. In addition, we present a checkpointing protocol that guarantees a \( d \)-bounded cycles execution.

Finally, we present a technique called overhead ratio for evaluating distributed checkpointing protocols. This technique gives a quantitative evaluation for each checkpointing protocol. Particularly, it allows us to compare various checkpointing protocols easily. As a result, we can choose the right protocol for a given environment.
List of Symbols and Abbreviations

\( n \) — Number of processes.
\( E \) — A distributed execution.
\( C_{p,i} \) — The \( i \)th checkpoint of process \( p \).
\( I_{p,i} \) — The \( i \)th checkpoint interval of process \( p \).
\( \rightarrow_{hb} \) — The Lamport’s happened before relation.
\( \rightsquigarrow \) — A zigzag path (Z-path).
\( R \) — A recovery line.
\( F \) — A cycle class.
\( IL(E) \) — The checkpoint interleaving level of execution \( E \).
\( \mathcal{P} \) — A distributed checkpointing protocol.
\( o \) — The overhead of a checkpoint.
\( L \) — The latency of a checkpoint.
\( r \) — The overhead of a recovery from a checkpoint.
\( \lambda \) — A failure rate of a process.
\( v \) — Overhead ration.
\( d\text{-BC} \) — \( d \)-bounded cycles
\( C/R \) — Checkpoint/Restart.
\( \text{OCVM} \) — Objective Caml Virtual Machine.
Chapter 1

Introduction

The world today is interconnected. That is, computers inside departments and organizations are connected by LANs, and these LANs are all connected to the Internet. Not surprisingly, a large body of research, as well as several commercial products, have addressed the issue of how to use as many connected computers as possible for carrying out large computations, utilizing these resources in an efficient way. This is done both at the LAN or cluster level, often referred to as cluster computing [15, 20, 59, 78], as well as at the Internet level, e.g., in the form of meta-computing [68, 71] and grid-computing [46].

Employing clusters of workstations as a cost effective alternative to parallel computers has been the goal of much research in the past few years [20, 36], especially in light of the remarkable advances in both computing power of PCs and networks speed. While the idea of building such clusters is very appealing, the realization of this goal is quite complicated, due to the numerous non-trivial issues that have to be dealt with. These include maintaining the promised performance at the application level, manageability of the cluster, and reliability. Other issues like job/process scheduling must also be considered, although these are well studied problems, and the usage of clusters vs. parallel computers does not significantly alter known solutions from the parallel world.

One major obstacle in the way of building such clusters, which has been successfully dealt with recently, lies in the fact that legacy operating systems present high overhead in accessing the network. Thus, naive usage of fast networks fails to deliver on the promised network speed. Fast-Messages [1], VI [11], NOW [20], U-Net [23], and other projects have been able to overcome this problem by developing user-level network interfaces, that bypass the operating system kernel, offering applications near optimal bandwidth and latency. Thus, largely speaking, obtaining low network latency and high bandwidth can be considered a solved problem (at least from the research point of view). However, the problems of achieving cluster manageability, high-availability and reliability without hurting the performance of applications under normal conditions are still far from resolved. In particular, the distributed nature of these clusters and the relatively high probability of partial failures that is inherent in distributed environments make these problems hard.

There are various approaches for dealing with reliability in distributed computing. Group Communication is one such approach. Group communication provides high-availability in distributed systems through replication [26]. Moreover, it can be configured to guarantee
totally ordered multicast among processes [26, 49]. While this approach has been applied with much success in control applications, trading and other fields, it is still far from being useful for high performance distributed computing systems.

Another approach for providing fault-tolerance and thus reliability is Checkpoint/Restart (C/R). Checkpoint is the act of saving an application’s state to stable storage (e.g., a disk) during its execution, while restart is the act of restarting the application from a checkpointed state. As depicted in Figure 1.1, if checkpoints are taken periodically, then when an application fails, it can be restarted from its most recent checkpoint. This limits the amount of computation lost due to a failure to the time that elapsed between the last checkpoint and the failure. Checkpointing can also be used for migrating a process from one computer to another, and for debugging purposes, etc. [106].

![Image: Periodic checkpoints for fault-tolerance.]

In this dissertation we focus on the problem of providing manageability, high-availability, and reliability in distributed systems without sacrificing the performance. We present Starfish\textsuperscript{1} [7], a system that tries to tackle this problem using a novel architecture that combines group communication and C/R technologies. Starfish as a system is manageable, highly available, and adjusts to dynamic changes in the cluster. For its applications, Starfish provides hooks to handle dynamic cluster changes, as well as C/R facilities. Specifically, Starfish is a collection of nodes (e.g., PCs) connected by a high speed network. The code of Starfish is written in OCaml [10], which is a virtual machine language. OCaml is fully portable to most variants of Unix, as well as Windows NT. Currently, Starfish supports both Myrinet [4] (for performance) and IP (for convenience).

Each Starfish node runs a daemon. The collection of these daemons form a group which uses the Ensemble group communication toolkit to maintain a replicated state of the cluster [9, 54]. These daemons are used to spawn the user applications. User applications in

\footnotesize\textsuperscript{1}Starfish is an asteroidea that lives in the sea. It belongs to the multiple class, and if one of its legs is cut off, it has the ability to “recover” it.
Starfish are MPI programs, which are of MDSP (Multiple Data Single Program) type.

Starfish architecture is very flexible and portable in the sense that it allows us to implement several different distributed C/R protocols and run them side by side. In particular, we can run the same application with two different C/R protocols and compare them. Moreover, Starfish supports both homogeneous and heterogeneous C/R. With heterogeneous checkpointing, Starfish becomes a heterogeneous system where its nodes can run on different architectures and operating systems. A heterogeneous C/R mechanism allows to restart an application from a saved state that was taken in a hardware architecture and/or operating system that can be different from those in the machine on which it is restarted.

Without any doubt, the problem of C/R is much more difficult in heterogeneous systems than it is in homogeneous ones. In the latter, a checkpoint can simply be done by dumping the process core. There are optimized implementations that attempt to greatly reduce the amount of saved data. But even they can rely on the fact that the architecture and operating system of the computer in which the failed application is restarted are the same as the ones in the computer in which the state was saved [81]. In particular, such implementations can assume that the data representation, the machine registers, the stack, heap and data segments, as well as the machine’s native instruction sets are all the same. Yet, in heterogeneous environments the above assumptions do not hold. Thus, to improve the utilization of clusters as well as meta and grid computing systems, it is desirable to construct C/R mechanisms that can operate across multiple platforms and operating systems.

In recent years there is a growing proliferation of virtual machine based programming languages, such as Java, C#, OCaml, and Lisp, that implement their own memory management. These languages are translated to bytecode representation, which is independent of a particular architecture’s instruction set. Similarly, the state of the application depends on the virtual machine’s internal registers, stack, heap and data segment representation, which are independent of operating system and hardware. Also, in order to support internal memory management, and in particular, garbage collection, data types are often tagged, which is useful when trying to interpret a block of saved data dumped in one architecture, and use it in another architecture. It is therefore conceivable to construct a heterogeneous C/R mechanism that will checkpoint the application’s state at the virtual machine level, rather than the native machine’s level, making the problem of heterogeneous C/R much easier.

In this dissertation we report on our experience, design decisions, methods, and optimizations for supporting heterogeneous C/R. We also present a performance analysis, and provide some thoughts and comments about how this work can be extended to OCaml native code and to Java Virtual Machine. In particular, we compare our work to recent work on mobile Java code, e.g., [28, 66]. Our comparison yields some aspects in which Java could improve to lend itself more easily to heterogeneous C/R. Also, note that work on agents and Java mobility require adhering to a unique programming model, and usually movable objects and threads must inherit from predefined movable base classes. On the other hand, our work does not assume anything about the programming model, and is completely transparent to the application. Thus, we can checkpoint and restart any application without modification, so our work is more suitable to numerical applications than agents.

Back to Starfish, since it is a distributed system, it needs to apply distributed check-
pointing protocols. However, the problem of C/R is more complicated in distributed settings, where an application is composed of several processes, each possibly running on a different computer. In order to restart such an application, one has to choose a collection of checkpoints, one from each process, that corresponds to a consistent distributed application’s state. A distributed application’s state is not consistent if it represents a situation in which some message $M$ is received by some process, but the event of sending $M$ is not in the checkpoint collection. A collection of checkpoints that corresponds to a consistent distributed state forms a recovery line. If a failure occurs when no collection of checkpoints taken during the execution forms a recovery line, then the application will have to be restarted from the beginning, regardless of the number of checkpoints already taken. The domino effect was identified in [87] as the source of not being able to find a recovery line in an execution with checkpoints, indicating that a recovery line is not guaranteed unless special care is taken.

In this dissertation we focus on the number of checkpoints that may need to be rolled back after a failure in order to recover from a recovery line. Having to rollback over a checkpoint is considered particularly bad, since the whole point of a checkpoint, and the reason to pay the performance penalty for taking it, is to limit the rollback. We define the notion of $k$-rollback, which indicates the maximal number of checkpoints a process may need to rollback during recovery. In addition, we introduce a new class of executions called $d$-bounded cycles ($d$-BC), based on our notion of a cycle, and show that $d$-BC is included in $(n - 1)d$-rollback, where $n$ is the number of processes in the system. Informally, an execution is $d$-BC if no checkpoint is included in a cycle whose length is more than $d$ checkpoint intervals.

We present a protocol that generates only $d$-BC executions for a given $d$. Our protocol does not piggyback any control information on messages. Control information is sent in special control messages and only after each checkpoint. Modern high speed networks, such as Myrinet [4], ServerNet [57], and even ATM, when used with a user level interfaces such as U-Net [23], Fast Messages [1], BIP [83], or VI [11], obtain latencies of less than 10 microseconds for short messages. Thus, protocols that need to piggyback control information on each message may in fact add a significant overhead when employed in such networks. Clearly, sending too many control messages is not desirable either. However, since our protocol refrains from doing both, we believe it is highly applicable to such ultra-fast networks.

An added bonus of our protocol is that each process maintains information about the most recent recovery line it is aware of due to control information. After a failure, it is possible to collect the last recovery lines known to each process. The union of these recovery lines is guaranteed to be the most recent recovery line, up to the message delivery delay of the network, that existed shortly before the failure occurred. Moreover, a process can garbage collect any checkpoint it has taken prior to the one included in its last recovery line without any additional communication and/or computation costs.

Lastly, we focus on evaluating distributed checkpointing protocols. In order for a checkpoint protocol implementation to be useful, the overhead it imposes on applications must be small. Otherwise, it would be preferable to run the application without checkpointing, and if a failure occurs, restart the application from scratch. In a uniprocessor system, the main issues that affect the checkpointing overhead are the speed of the media to which the state is saved, whether it is a disk or a remote computer [79], the size of the data that needs
to be saved and the ways to retrieve it. For example, a common technique to reduce this size is to perform incremental checkpointing, in which the full state is rarely saved, and only the changes that were made in between full checkpoints are logged on stable storage [81]. For parallel or distributed applications, the situation is more complicated. As we indicated before, in such an environment, if each process saves its state in a completely independent manner, it is possible that a recovery line will not form. Thus, the main research focus in this area is on devising techniques that guarantee the existence of a recovery line while minimizing the coordination between processes. This coordination overhead includes the amount of control information exchanged between processes and the number of times some process $p$ is forced to take a checkpoint to ensure that a recovery line exists. Such checkpoints are called forced checkpoints.

Over the years, a multitude of C/R techniques have been proposed for both the single processor and the parallel/distributed settings. This has created a need for objective quantitative evaluation measures that allow one to compare the various approaches on the same scale. One such scheme has been developed by Vaidya in the context of applications running on a single processor [98], but it does not extend trivially to most distributed C/R mechanisms. There are two reasons for this. First, Vaidya’s work does not take into account the communication costs associated with distributed checkpointing, and second, it assumes that each process always restarts from its most recent checkpoint. As we show in this work, this assumption cannot usually be made in distributed C/R schemes, since these schemes often require a process to rollback after a failure to a more distant checkpoint [13]. In this dissertation we generalize Vaidya’s framework to parallel/distributed checkpointing schemes, by incorporating into it issues of communication overhead and rollback. In particular, we show that when a checkpoint mechanism belongs to the 0-rollback class, our evaluation measure agrees with Vaidya’s results.

The organization of the remainder of this thesis is as follows. Chapter 2 discusses related work. This chapter compares Starfish with other existing distributed system in order to give the reader a feeling of Starfish innovation and contribution. Then it surveys heterogeneous checkpointing work and presents the main approaches to perform distributed checkpointing. Chapter 3 presents the system model used in our work and then it presents a framework for defining rollback-recovery in distributed systems. Chapter 4 presents the Starfish design and architecture. This chapter describes the various components of the architecture, and illustrates their interactions by events. Then, it shows initial experimental results of Starfish performance. Chapter 5 describes the implementation of heterogeneous checkpointing in the virtual machine language OCaml. Moreover, it discusses how to apply our implementation in other virtual machine languages like Java and C#. Chapter 6 discusses the main issues regarding distributed checkpointing. It presents the notion of a cycle and classifies distributed executions based on the maximum propagation of rollback for recovery upon failures. Then, it presents the new class of $d$-BC and presents a checkpointing protocol ensuring $d$-BC. Chapter 7 presents a measure for performance analysis of distributed execution based on all the important parameters affecting the execution. It presents a technique of computing this measure and how to use it for comparing distributed checkpointing protocols. Lastly, Chapter 8 summarizes the thesis and presents potential future work.
Chapter 2

Related Work

2.1 Overview

There are several systems that offer fault-tolerance by applying C/R capabilities, e.g., Condor [68], Manetho [43], and LoadLeveler [2]. Generally, C/R protocols can be categorized as coordinated, in which case all processes coordinate their checkpointing to form a global consistent state [34, 44, 81], as uncoordinated, in which case every process can perform checkpointing independently [81, 87, 105], or as communication induced checkpointing, in which case every process can perform checkpointing independently and can take forced checkpoints under a pre-defined conditions for ensuring consistent states [44]. One of the important aspects of Starfish’s architecture is that it enables us to implement and study all three methods within a single framework.

In this chapter we compare Starfish with other existing distributed systems. We emphasize the differences between Starfish and those presented systems. One of the important features of Starfish is providing heterogeneous checkpointing. We present related work on heterogeneous checkpointing and illustrate the novelty of our approach of providing heterogeneous checkpointing in the virtual machine level. Moreover, we survey some related work on distributed checkpointing and present existing research in the area of performance analysis for checkpointing protocols. Lastly, we discuss replication schemes as an alternative approach for fault-tolerance in distributed systems.

2.2 Comparing with Existing Systems

Starfish supports several approaches of C/R and employs group communication for providing fault-tolerance and high availability. In addition, Starfish allows dynamic changes in the number of running processes. Most of these features and other properties of Starfish have been studied. Thus, there are several distributed systems, in academia as well as in industry, that have some of these features.

Given the large number of high-performance distributed computing systems built, it is impossible to mention all of them. Here we discuss the ones we feel are more related to our work, and examine their architecture and functionality compared to Starfish.
Condor [68] is a distributed system that runs on a cluster of workstations. Condor provides an environment for executing serial and parallel applications on clusters. Moreover, it supports C/R in order to provide fault-tolerance and process migration [70]. Condor employs the sync-and-stop protocol [80] and imposes several restrictions on C/R in programs (for more details see Section 2.3). Moreover, Condor supports only homogeneous checkpointing in Unix-like systems [85]. On the other hand, Starfish supports both homogeneous and heterogeneous checkpointing. In addition, Starfish architecture allows us to implement, side-by-side, different approaches of checkpointing. Also, we believe that using the Starfish architecture we can remove most of the restrictions imposed in Condor [16].

Manetho [43] is a distributed system that runs on a cluster of workstations. It uses a novel combination of rollback-recovery and process replications to provide fault tolerance and high availability; Manetho uses coordinated checkpointing protocol as described in [34] and uses process replication to provide high availability to servers in the system [26]. Manetho employs only homogeneous checkpointing, while Starfish supports both homogeneous and heterogeneous checkpointing. Aside from supporting different approaches of checkpointing, Starfish high availability mechanisms are somewhat different than Manetho, as we use a group communication system to manage our cluster.

Libckpt [82] is a transparent checkpointing library for uniprocessors running UNIX. Libckpt implements most optimizations that have been proposed to improve the performance of checkpointing, e.g., incremental checkpointing, forked checkpointing, copy-on-write checkpointing [81]. However, libckpt is merely a library, whereas Starfish is a complete system that exports a transparent checkpointing library [17]. Also, libckpt does not address high availability and dynamicity as Starfish does.

LoadLeveler [2] is a distributed system that runs on a cluster of workstations. It provides an environment for executing serial and parallel applications with dynamic scheduling. In addition, it supports C/R only with serial jobs in order to balance workload and provide process migration, and is thus incomparable to Starfish which is dedicated mainly for executing MPI programs.

Legion [50] is an object-based meta-system. It is built on a collection of connected hosts to provide a virtual computer that can access all types of data and physical resources. Legion is designed to be a worldwide virtual computer, while Starfish is designed to be a reliable and highly available distributed system for executing message-passing applications on clusters.

HPVM [36] is a distributed system that runs on a cluster of PCs with Windows NT. This system achieves high performance communication by using modern processors (300 MHz Pentium II) and FM protocol for communication on Myrinet [19, 4]. In addition, the system includes efficient implementation of standard scientific computing APIs such as MPI [3]. However, HPVM does not support fault-tolerance or high availability. In addition, Starfish achieves high performance communication by using Myrinet with the BIP protocol [8].
Millipede [48] is a Distributed Shared Memory (DSM) system that runs on a cluster of workstations. It supports various consistency models of DSM, as well as thread migration inside the cluster for load-sharing and to improve the locality of memory references. Millipede does not support fault-tolerance, parallel I/O, and security [59]. Starfish, on the other hand, supports process migration for load-balancing and fault-tolerance [81]. Moreover, Starfish is a message-passing system where Millipede is a DSM.

NetSolve [5] is a client-agent-server system. NetSolve allows a client to access computational servers across the network using a remote procedure call (RPC). In addition, NetSolve supports fault-tolerance by applying C/R on the server side [17]. Starfish, on the other hand, is not a client-server system, it is just a cluster of workstations connected by high speed networking for running high performance applications.

Aurora [32] is a cluster of processors that are completely connected. It supports C/R for capturing a partial state of the application according to the user specification. The system generates code for applying such C/R. Furthermore, Aurora does not support distributed C/R. Starfish, on the other hand, supports transparent distributed C/R.

Symphony [47]. It is a management infrastructure for executing virtual servers on the Internet. Symphony is based on combining CORBA and group communication technologies. Symphony is not a high performance distributed system. It provides hooks for server dynamic changes on the Internet. Moreover, it does not support fault-tolerance for the servers.

2.3 Heterogeneous Checkpointing

There are two main approaches for performing a local checkpoint, i.e., saving the local state of a process. One approach is to implement C/R in user space such as in Starfish (see Appendix A), Condor [68], CLIP [35], and Libckpt [82]. Implementing C/R in user space imposes a set of restrictions. There are many checkpoint implementations that do not support signals, timers, memory-mapped files, or shared memory since they are usually part of the kernel of an operating system [69]. The second approach is to implement C/R in kernel mode, such as in Unicos [63] and Sprite [77]. Implementing C/R in kernel space eliminates most of these restrictions, but requires modifying the Kernel.

Most published work regarding C/R addressed homogeneous systems. To the best of our knowledge, there are only a handful of research papers dealing with C/R mechanisms for heterogeneous systems. Most of them are based on compiler intervention and adding explicit calls in the code. We briefly describe these papers below.

Ferrari et al. [45] have developed an approach to checkpoint the dynamic state of processes in a platform-independent manner called Process Introspection. This approach centers on the semantic-preserving modification of programs by a source code translator that incorporates C/R functionality into processes. Another approach of C/R in heterogeneous systems was proposed by Theimer and Hayes [94]. In their proposed approach, the state of a process is
examined and checkpointed using compiler-generated symbol mapping information. Moreover, there are other approaches that deal with process migration on heterogeneous systems. For example, Symphony has a proposed mechanism for server migration on the Internet [47].

In addition, automatic program-transformation mechanisms have already been used to obtain heterogeneous C/R. Those mechanisms based on translating a program’s code to another in which new calls are added and define explicit data structure for performing heterogeneous C/R. Porch and c2ftc are C-to-C compilers that translates C programs into semantically equivalent C programs that are capable of saving and recovering from portable checkpoints [6, 86].

### 2.3.1 C/R in Java and Java Mobility

As we show latter, we implemented heterogeneous C/R inside the virtual machine of OCaml [10]. When considering Java, our work is highly relevant, since any work that attempts to do heterogeneous C/R in a JVM would have to tackle the same issues that we did in Starfish, as presented in Chapter 5. Also, since Java is also a strongly typed language that relies on a garbage collector, our solutions can probably be adapted to Java as well. On the other hand, there are several different JVMs, which might have different internal representations and each might manage its memory segments differently. Thus, to support cross JVM C/R one might need to be able to translate these concepts, which starts to resemble heterogeneous native code C/R. Additionally, the use of Just-In-Time compilation is now common practice in most JVMs. This complicates the task since the mechanism would need to support both native code and bytecode. In particular, parts of the same application might run inside the JVM while other parts might run as native code, with cross dependencies between them. Finally, another complication with Java is the fact that many JVMs allow using native threads when available, unlike OCaml that always uses its own threads.

To the extent that we are aware of, research on C/R in Java and Java mobility is mostly targeted towards moving agents, e.g., [66], rather than heterogeneous C/R. This has resulted in several differences between our approach and what has been done for Java. For example, Lawall and Muller [67] suggested a checkpoint mechanism in which the state of a Java application can be restarted from the contents of the object fields. They require that each object invoke particular methods from a predefined interface called Checkpointable. Then, in the implementation level, they perform a serialization for these objects.

Some projects, e.g., JavaGo [89] and Brakes [96], implement C/R at the language or bytecode level using a post-processor. These projects post-process the application’s bytecode in order to facilitate saving the threads stacks in order to enable mobility.

Bouchenak [28] suggested a mechanism that enables checkpointing the state of a running thread in Java and restarting this thread on a possibly different machine, both for persistence and mobility. Note that the state of a Java thread consists of three parts: the heap, the method area, and the stack. The heap and method area consist of information that is portable across heterogeneous architectures, but not necessarily across different JVMs, while a Java stack includes data structures in their native representation. A four-byte word in the

---

1 Of course, one can always disable the use of the JIT compiler and avoid these problems.
stack may represent either a Java reference, an int value, or a float value. On the other hand, a four-byte word in an OCaml stack represents either an OCaml reference or an int value. A float value in OCaml is a blocked type and is placed in the heap. In addition, Bouchenak suggests two approaches for recognizing whether a value in the stack is a reference or a native type. In the first approach, recognition at runtime, the type information is actualized each time a bytecode instruction is interpreted by the JVM. The obvious drawback of this approach is the induced overhead on the application performance. In the second approach, recognition at capture time, the type information is analyzed during checkpointing. This approach induces an overhead on checkpointing, which could be prohibitively high, if the stack contains a big list of frames. It seems that Java could have lent itself more easily to C/R at the JVM level if it were using a similar approach to OCaml that simplifies the distinction between references and native values.

Furthermore, since Bouchenak is interested in thread migration and persistence, rather than complete applications, only objects that are accessible by a given thread need to be checkpointed in [28]. Thus, the mechanism suggested by Bouchenak must scan all such objects and serialize them one by one. In comparison, our mechanism simply dumps the heap in its native representation, and only if necessary, translates the saved heap to the restarting machine’s representation during restart. When an entire Java application needs to be checkpointed, rather than just a single thread, one could use our technique of dumping the heap as is. However, since non-serialized objects’ representation is not standardized, this might not work across virtual machines from different vendors. It would be beneficial if the Java standard would require that the structure of an Object in the heap have enough information to be parsed in a generic way by different virtual machines.

The heterogeneous C/R proposed in [28] requires the application to use the APIs of the ThreadStateManagement class while thread migration requires the use of the MobileThreadManagement class. Unlike our approach, this heterogeneous C/R service is non-transparent in the sense that the user must use a set of APIs explicitly and then compile the program with the appropriate packages.

Finally, the Merpati system [93] also implements a C/R mechanism at the Java virtual machine level. Merpati exposes an API that allows applications to request a checkpoint or migration to another machine. One of the main differences between Merpati and our work is that Merpati does not fork the process, and therefore the application is blocked for the entire duration of the checkpoint. Moreover, Merpati requires all threads to reach a safe point and stop there on a conditional variable, where the requirements from a safe point in Merpati include not running in a JIT frame. If some thread is stuck on a semaphore and does not reach a safe point after some timeout, the checkpoint is aborted. Given Java’s representation of objects, a large part of the effort in Merpati goes into analyzing in run-time the class files in order to figure out how to save and restore objects’ data.

2.4 Distributed Checkpointing

As we indicated in Chapter 1, there are three main approaches for performing a checkpoint in a distributed system for message-passing applications. The first approach is coordinated
checkpointing, where processes coordinate the checkpoint operation. Whenever one process wishes to take a checkpoint, all processes coordinate their actions and take a checkpoint as well. The two common techniques for coordinated checkpointing are either to synchronize and pause the execution of all processes until everyone has taken a checkpoint [44, 81], or to use Chandy-Lamport’s distributed snapshot protocol [34]. The second approach is called uncoordinated checkpointing, where each process takes a checkpoint independently of other processes. Every node takes a local checkpoint whenever it wishes to, without any coordination with the others. This, of course, does not impose any overhead, but runs the risk of never generating a recovery line [44, 104]. The third approach is called communication induced checkpointing, in which processes take local checkpoints in an uncoordinated manner. However, occasionally, based on the communication patterns, a process may be forced to take a checkpoint in order to guarantee the existence of a recovery line [55, 97]. Such checkpoints are called induced (or forced) checkpoints. There are several techniques for improving the performance of this approach such as the optimistic recovery mechanism, where processes are still taking their local checkpoint asynchronously [92]. Much of the work on communication induced checkpointing identifies the necessary and sufficient conditions for having a recovery line, and develop protocols that force a process to perform a local checkpoint whenever some of these conditions might be violated. Most notably, the existence of a checkpoint within a Zigzag-path (Z-path), and, in particular, a Zigzag-cycle (Z-cycle), was identified as the source for not having a recovery line in an execution [18, 75]. The corresponding protocols force a process to perform a local checkpoint whenever there is a fear that by not taking a checkpoint, some other checkpoint could become part of a Z-path or a Z-cycle [21, 56, 108].

Other approaches for reducing the rollback propagation include taking a checkpoint after each send [107, 104], smart message scheduling techniques [105], and the message logging technique [80, 81]. The latter approach assumes that program executions on each processor are piecewise deterministic, and each processor logs all received messages. Rao et al. [88] evaluated the cost of recovery in this approach.

Manivannan and Singhal [73] classified executions by the degree to which the creation of various Z-paths that include checkpoints are prevented, and identified the following classes: strictly Z-path free (SZPF) includes executions in which there are no non-causal Z-paths between any two checkpoints; Z-path free (ZPF) includes executions in which for every Z-path there is a corresponding causal path; Z-cycle free (ZCF) includes executions in which no checkpoint is inside a Z cycle. Manivannan and Singhal also showed that $SZPF \subseteq ZPF \subseteq ZCF$. The intuition is that a protocol that enforces a more refined condition also applies fewer forced checkpoints.

In Chapter 6 we extend the work of Manivannan and Singhal by suggesting a new classification of C/R protocols. We show that checkpoint-after-send is 0-rollback, coordinated checkpointing, SZPF, and ZPF are 1-rollback, while ZCF is $n$-rollback and is a special case of $d$-BC for $d = 1$. At the top of the classification is unbounded-rollback, which we show to be equivalent to uncoordinated checkpointing.\(^2\) Our protocol forces fewer checkpoints than stricter properties such as SZPF, ZPF, and coordinated checkpointing.

\(^2\)We consider an execution without any checkpoints as an uncoordinated checkpointing execution as well.
2.5 Performance Analysis

There has been much work on checkpointing performance analysis for both uniprocessors and distributed systems [74, 84, 101, 99, 110]. Most of these papers did not take into account the rollback propagation. Our work is the first to incorporate all the parameters which affect the performance in distributed environments into an analytical measure.

Mishra and Wang [74] evaluated several checkpointing protocols by implementing them and then running them with test application for obtaining their overhead.

Ziv and Bruck [110] compared only four checkpointing protocols by using the Markov Reward Model [95]. For each checkpointing protocol, they defined a particular Markov chain depending on the number of failures that can occur simultaneously during the execution. Our approach differs from [110] in that we provide a technique for comparing any checkpointing protocol based on its rollback propagation. Moreover, Ziv and Bruck presented in [111] a checkpoint scheme for duplex systems. Such a system is formed by a pair of processors connected by a LAN. The state of the two processors must be compared to detect a failure. The comparison is performed on the local checkpoints of the processors. Ziv and Bruck conduct a performance analysis for their scheme in the duplex system, but it is not a general system for distributed executions.

Vaidya defined the overhead ratio for uniprocessor systems as a function of the checkpoint overhead and latency [98]. He proved that the optimum checkpoint interval depends on the value of the local checkpoint overhead. In addition, Vaidya claimed that the overhead ratio can be computed for distributed systems as in uniprocessor systems by taking the values of parameters, such as the checkpoint latency, either to be the maximum or the average over all processes. In another work [100], Vaidya computed the overhead ratio for the two-level recovery approach. This approach tolerates single failures with a low overhead and multiple failures with a higher overhead.

Plank and Thomason [84] have presented a method for estimating the overhead ratio for distributed executions with coordinated checkpointing. By assuming coordinated checkpointing, they do not address rollback propagation. Moreover, they do not address the control overhead incurred by control information.

2.6 Fault-Tolerance by Replication

The other major technique for providing fault-tolerance is by replication. There are two approaches for replication: primary/backup [31, 30, 52] and active replication, also known as replicated state machine [90]. In primary/backup, one node, known as the primary, performs the computation and occasionally dumps its state either to a shared non-volatile storage, or over the network to the backups. If a failure occurs, one of the backups takes over, and resume execution based on the state available to it. In active replication, all update requests are delivered to all replicas in the same order, and each applies the updates locally in that order. When update request execution is deterministic, this ensures that all replicas continuously hold the same state. Figure 2.1(a) presents an example of how the primary acts upon a client request and Figure 2.1(b) presents an example of an active replication scenario where
the client waits until it gets the first response [52].

![Diagram of primary/backup and active replication approaches.](image)

Figure 2.1: Examples of primary/backup and active replication approaches.

Implementations of primary/backup largely vary in the guarantees they make about the recency of the data available to the backups. In some applications, the information available to the backups might not be the most up to date. This is analogous to C/R, with the difference that in primary/backup typically only the logical state of the application is saved and not the entire process state. In particular, implementations that propagate the entire logical state can be compared to naive checkpointing schemes, where implementations that only propagate the results of the last update(s) can be compared to *diff checkpointing* [81].

When the application requires that the backup recovers from exactly the same state the primary held just before crashing, this calls for a hand-shake protocol between the primary and the shared disk, or the primary and the backups. An example of the latter are 2-phase commit protocols [24].

There are two common approaches to implementing active replication, with some overlap between them. The first approach is based on *consensus* as a building block. That is, each replica that wishes to initiate an update, runs a consensus protocol to determine what is the next update, and all replicas perform the decided update request. The consensus protocol is often a variant of Paxos [65], the Chandra-Toueg algorithm [33], or the extended three phase commit protocol [62], all of which are essentially equivalent. The second approach is based on group communication and virtual synchrony [26, 102]. The main difference between the two approaches is that group communication based solutions already include inherent handling of dynamicity. Also, group communication allows various levels of guarantees, namely, partitionable view synchrony, primary partition, and safe delivery. These enable the application developer to chose between several levels of the performance vs. consistency tradeoffs within the same framework.

The distributed atomic transactions model is also used to provide fault-tolerance in data-bases and file systems [24]. By exploiting the semantics of transactions, that model allows for some concurrency between non-conflicting transactions. Also, it enables coordination in an environment that is not fully replicated, and allows parties to object to an update. Atomic transactions also require running an agreement protocol, such as the 2-phase commit [24], 3-phase commit [91], or extended 3-phase commit [62].

Lastly, quorum based replication allows protocols in which only a subset of the replicas participate in each operation [72]. These schemes are especially useful in allowing clients to interact directly with the replicated data, without passing though mediating servers [37].
Chapter 3

The Basic Framework

3.1 The System Model

We consider a distributed system consisting of \( n \) processes, denoted \( P_1, P_2, \ldots, P_n \), connected by a communication network and communicating by exchanging messages asynchronously over the network. Processes are modeled as automata, starting in some predefined state, and each executing multiple steps. In each step, a process receives zero or more messages, performs some computation, and generates zero or more messages to be sent to other processes. Additionally, in each step a process may decide to take a checkpoint, or to perform restart. A restart simply sets the state of the process to a state that was saved in a previous checkpoint. Finally, a process may fail by crashing during any step. A process that crashes does not take additional steps until it performs a restart. On the other hand, a process may perform restart even if it did not crash, but another process did. We refer to sending/receiving a message, taking a checkpoint, and crashing as events that occur in the process.

We define a local history of a process to be the sequence of events and state transitions it incurs. An execution is a collection of histories, one for each process, which obeys the typical well-formedness properties. For example, a message cannot be received if it is not sent in the execution. In addition, we assume that the network delivers messages reliably in First-In-First-Out (FIFO) order [34]. For each receive event in an execution there is one corresponding send event, and for each send event there is at most one receive event. Moreover, if the execution is infinite (i.e., there is a process which has an infinite number of events in its local history), then for each send event there is exactly one corresponding receive event.

Events in an execution are related by the happened before relation [64] denoted \( \text{hb} \); this relation is defined as the transitive closure of the process order and the relation between the send and receive events of the same message.

We assume that there is a global clock that may not be accessible to the processes. Each event in a history is associated with a global time at which this event occurred; this association is expressed by a time function \( t(event) \). This function is monotonically increasing for events in a local history. The time at which a message is received is later than or at the same
time at which it is sent; in other words, the time function is consistent with the happened before relation.

Each checkpoint taken by a process is assigned a unique sequence number. The $i$th checkpoint of process $p$ is denoted $C_{p,i}$. The $i$th checkpoint interval of process $p$, denoted $I_{p,i}$, is the sequence of all events performed in the interval $[t(C_{p,i-1}), t(C_{p,i})]$. For instance, in Figure 7.1(a), the checkpoint interval $I_{p,i}$ contains all the events performed from $C_{p,i-1}$ until but not including $C_{p,i}$. To simplify the presentation and discussion, we use the following convention which slightly abuses the above notation. That is, when a process $p_i$ performs restart from a state that was saved during checkpoint $C_{p,i}$, we denote $p_i$’s next checkpoint (assuming no additional failures and restarts) by $C_{p,i+1}$.

We assume that process failures occur randomly and independently under exponential distribution with rate $\lambda$.

### 3.1.1 Practical Considerations

In practice, a process typically consists of several modules as depicted in Figure 3.1. Namely, the Application module executes the user program; the Checkpoint/Restart (hereafter, C/R) module executes the checkpointing and recovery protocols (for coordinating taking checkpoints and recovery with other nodes); the Checkpointer component performs the checkpoint (saving the local state); the System Layer represents the interface between the process and the operating system.

This model allows us to make the following distinctions: Messages generated by the application module are considered *data messages* while messages generated by the C/R module are called *control messages*. Moreover the C/R module can intercept data messages and piggyback on them *control data*. The combination of control messages and control data in the *control information* used by the C/R protocol.

Similarly, when the checkpoint decision is made by the C/R module, we call it a *forced checkpoint*, reflecting the fact that it was taken due to the protocol. Otherwise, it is an *independent checkpoint*, reflecting the fact that it was taken due to local considerations only.
3.1.2 Distributed Checkpointing

When a failure occurs in a distributed system, we need to recover from a cut of checkpoints (i.e., a set of checkpoints consisting of one checkpoint for each process). However, not all cuts of checkpoints are consistent, i.e., correspond to a state that could have been reached in the execution. A consistent cut of checkpoints is called a recovery line, and is characterized by the following definition.

**Definition 3.1.1:** A cut of checkpoints $S$ is a recovery line if for each receive event recorded in $S$, the corresponding send event is also recorded in $S$. 
Chapter 4

The Starfish System

4.1 Overview

In this chapter we present the architecture and design of the Starfish system, which is an environment for executing MPI programs on a cluster of workstations. Starfish tries to tackle the features of reliability and manageability by using a novel architecture that combines group communication and C/R technologies. Starfish as a system is manageable, highly available, and adjusts to dynamic changes in the cluster. For its applications, Starfish provides hooks to handle dynamic cluster changes, as well as C/R facilities. The initial implementation of Starfish runs on Linux, and supports both Myrinet [4] (for performance) and IP (for convenience).

Most of Starfish code is written in OCaml [10] (see Section 5.2). OCaml allows system developers to leverage powerful language support to attain high performance and flexibility. On the other hand, as discussed in Chapter 5, since OCaml supports bytecode, it helped us make Starfish a heterogeneous reliable system that may run on any collection of platforms. Furthermore, Starfish is fully portable to most variants of Unix, as well as Windows NT. The only part that needs to be rewritten for these operating systems is adapting the VNI (see Figure 4.2) code for Myrinet (or other fast networks) to these systems.

As depicted in Figure 4.1, each Starfish node runs a Starfish daemon (or simply a daemon). All Starfish daemons form a process group [26], using the Ensemble group communication toolkit [9, 54]. As described later, these daemons are used to interact with clients, spawn MPI programs to which we refer to as application processes, track and recover from failures, and to maintain the configuration of the system. In particular, daemons utilize a lightweight group mechanism, in a manner similar to the group daemon proposed in [27], for keeping track of and reporting partial application and system failures.

Another interesting feature of Starfish relates to its API. The additional functionality of Starfish is supported through additional downcalls and upcalls. For each of the upcalls, there is some default handling procedure, and hence applications that do not wish to handle these upcalls can simply ignore them. Similarly, applications are not required to issue any downcalls. This allows Starfish to run regular MPI programs without any modifications. Naturally, such programs will only enjoy part of Starfish’s capability, e.g., system initiated
checkpointing, but not all the potential benefits of the system, e.g., user initiated checkpointing and dynamic reconfiguration. Conversely, programs that use the additional API calls can be automatically, or semi-automatically, transformed back into standard MPI programs by eliminating all Starfish specific downcalls. Since these calls only deal with checkpointing and reconfiguration, such programs will then run correctly on any standard MPI implementation.

4.2 Starfish Architecture

As illustrated in Figure 4.2, each Starfish node runs a daemon, where all Starfish daemons are members of the same process group, called the Starfish group. This group is managed by the Ensemble group communication system [9, 54]. The collection of these daemons form Starfish’s parallel environment, and they are responsible for spawning application processes, keeping track of applications health, managing the configuration and settings of the cluster, communicating with clients, and for providing the hooks necessary to provide fault tolerance for applications. In particular, all configuration and control messages, including those related to applications (but not data messages) are sent by the daemons using Ensemble.

Given the parallel nature of MPI programs, each application is expected to be divided into several concurrent processes, each potentially running on a separate node. As described shortly, each application process consists of a Starfish run-time environment and user supplied code. Starfish run-time library is responsible for interacting with the daemons, for checkpoint and restart, and for implementing MPI, whereas the user code is any given MPI program. We assume that the stable storage to which a process saves its checkpoints is accessible to all other nodes.

For instance, Figure 4.1 presents an example of Starfish with two MPI applications, where one application uses Ethernet and the other Myrinet for message passing.

---

1Ensemble ensures reliable ordered message delivery, as well as consistent automatic failure and recovery detection.
4.2.1 Daemon Internals

Starfish daemons need to maintain some application specific data for each application processes running on the same machine, as well as some shared state that defines the current cluster configuration and settings. We maintain this data by employing the design illustrated in Figure 4.2. Daemons are composed of four main modules: Ensemble (the group communication system we use), the management module, the lightweight membership module, and several instances of the lightweight endpoint module [53]. Daemons may exchange control messages among themselves; these messages may be generated by either module, and are sent through Ensemble, but are not passed to application processes (See Table 4.1).

In our design, a single management module is used to maintain the configuration and shared settings of the cluster. Each application that runs on Starfish is associated with a lightweight process group, whose members are the daemons on the nodes that run the corresponding application processes. For each application process, we instantiate a lightweight endpoint module. This module is the one that is responsible for the connection with the application process, and for passing messages and events to and from that process. The lightweight membership module is responsible for deducing the lightweight group membership from the entire Starfish process group, and for translating membership and message events between the main Starfish group and each of the lightweight groups.

Figure 4.3 presents an abstract view of Starfish with several lightweight groups. Here, all daemons are members of the same Starfish group. Additionally, daemons $p_1$, $p_2$, and $p_3$ share the same lightweight group, indicating that there is an application that spans all three machines. Similarly, $p_3$ and $p_4$ share a lightweight group, etc. Note that $p_8$ does not appear in any lightweight group, indicating that no application process is currently running on the corresponding machine.

Typically, the lightweight groups are only subsets of the main Starfish process group. This is because on a large cluster, each application spans less than the entire cluster. Thus, not every change in the Starfish group needs to propagate to every lightweight group. Similarly, if an application process on one node terminates, but the machine itself continues to run, then this should result only in a membership change of the corresponding lightweight group, and need not be reported to all members of the main Starfish group, or in other lightweight groups. Also, messages that are sent inside a lightweight group should only be delivered to
members of the same lightweight group.

Of course, it would have been possible to allocate a separate full blown process group for each application. But as indicated in [53], the lightweight group approach is more efficient. Also, the structure we described allows us to maintain both consistent cluster wide information, which is independent from a specific application, and manage multiple applications on top of this cluster, mimicking a parallel computer.

### 4.2.2 Application Process Internals

Each application process is composed of five major components, as illustrated in Figure 4.2: A group handler, which is responsible for communicating with the daemon, an application part, which includes the user supplied code, a checkpoint/restart module, an MPI module, and a virtual network interface (or VNI for short). All modules communicate by posting events on an object bus, that invokes the corresponding event handlers at each of the listening module. Using an object bus allows us to completely decouple the modules, and also to potentially post the same events to more than one module. Finally, in order to orchestrate these modules, we have implemented our own scheduler.

Application processes are involved in five types of messages: data messages, coordination messages, C/R messages, lightweight membership messages, and configuration messages (See Table 4.1). There is a significant difference between data messages and other types of messages: Data messages result from the user supplied MPI code and have strict performance requirements, while other messages are generated by Starfish itself and do not require the same responsiveness as data messages. Thus, we employ a fast data path between the MPI implementation and the application module that does not go through the object bus. This ensures the required low latency for data messages, while still being able to provide manageability and strong guarantees for the other types of messages.

To send other types of messages, the generating module posts an event for the group handler module and the group handler translates this event to a message on the TCP con-
nection with the daemon. In the case of coordination messages and C/R messages the daemon broadcast them in the relevant lightweight group using Ensemble. In the other direction, messages received by the group handler module are translated to events on the object bus, to be invoked at the corresponding modules in the application process. Using the daemon, and therefore Ensemble, for disseminating coordination messages greatly simplifies our code, and enjoys the strong delivery guarantees of Ensemble, which also simplifies our protocols.

Lightweight membership messages and configuration messages are part of the protocol executed between the application processes and the daemons, and are discussed in the next subsection. Coordination messages are sent among application processes (potentially) located on different nodes for general coordination tasks, while C/R messages are used by the various C/R protocols. The set of C/R messages seems to be rich enough to express all C/R protocols we have encountered. Since lightweight membership messages and C/R messages are exchanged by application processes, and daemons only serve as a reliable middle communication layer, these messages are opaque to daemons.

<table>
<thead>
<tr>
<th>Message type</th>
<th>Sent between</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Starfish daemons</td>
</tr>
<tr>
<td>Coordination</td>
<td>Application processes through daemons</td>
</tr>
<tr>
<td>Data</td>
<td>Application processes through MPI and VNI modules using fast path</td>
</tr>
<tr>
<td>Lightweight membership</td>
<td>Lightweight endpoint module and application processes</td>
</tr>
<tr>
<td>Configuration</td>
<td>Local daemon and application processes</td>
</tr>
<tr>
<td>Checkpoint/Rstart</td>
<td>Checkpoint/restart modules through daemons</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of message types in Starfish

Optimizing Receives

As in the MPI standard, Starfish supports both blocking and non-blocking send and receive operations. In blocking mode, a send or receive operation does not return until the message has been fully sent or received, while in non-blocking, operations return immediately. The eager way of implementing sends is to immediately send a message to its destination [51]. This, however, requires that the destination be prepared to read a message from the network, even if the application process there has not yet performed a matching receive operation.

In Starfish we overcome this problem by introducing a low priority thread, called the polling thread. This thread continuously polls the network, so whenever a message arrives,
the polling thread receives the message and puts it in a queue of received messages, for further handling by the application at a later time.

A nice feature of the polling thread is that it eliminates much of the runtime overhead of issuing a receive operation at the application level. A receive operation, particularly in blocking mode, pauses the process execution in order to receive an in-transit message. Moreover, when using the regular TCP/IP stack, receiving a message from the network involves a system call and user-level/kernel interaction, which is costly. When using the polling thread, the time required for kernel interaction is interleaved with other operations, yielding fast receive operations.

4.2.3 Interaction Between Daemon and Application Process

As mentioned in the previous subsection, two types of messages are designated for the interaction between daemons and application processes. These are configuration messages and lightweight membership messages. Lightweight membership messages inform application processes about new views, and are used by an application process to terminate its membership in a lightweight group. Configuration messages, on the other hand, are used to inform the application process of various configuration parameters, and used to synchronize between the application process and the daemon upon initialization and termination of the application process.

Actually, the interaction between an application process and its corresponding daemon is performed via the TCP connection between the group communication handler module of the process and its corresponding endpoint module in the daemon. Figure 4.2 illustrates such interaction between these modules.

4.3 Starfish Features

In this section we elaborate on how we achieve cluster manageability and high-availability, how we support dynamic changes in the environment at both the clusters and the application, and provide fault-tolerance for the application. We split the discussion into general cluster aspects, which are reported in Subsection 4.3.1, and to application aspects, which are reported in Subsection 4.3.2.

4.3.1 Starfish Manageability, Dynamicity, and High Availability

Manageability

Starfish can be managed from any computer connected to the LAN on which the cluster runs, either directly or through the Internet. Managing the cluster is done by opening a TCP connection to one of the daemons on which an ASCII based protocol is used. Through this connection, the cluster administrator can add or remove nodes from the cluster, disable and (re)enable nodes, and control the parameters of the cluster. The management protocol starts
with a login session, in which the client side has to authenticate itself as an administrator
to the cluster, and identify the connection as a **management connection**.

The management module of Starfish handles management connections, and takes care of
forwarding configuration commands to all other daemons in the system. The use of Ensem-
ble’s reliable and totally ordered delivered mechanism is instrumental here, in maintaining
coherent state between all cluster daemons [26].

A similar protocol that also employs a TCP connection is used between clients and any
of the cluster nodes in order to submit applications for either interactive or batch execution.
This protocol also begins with a login session, but is identified as a **user session**, and is thus
limited to submitting, suspending, resuming, and deleting applications. (A user can only
suspend, resume, and delete its own applications.)

Note that user commands regarding an application have to be propagated to all daemons
that manage the corresponding application processes, and in some cases should be forwarded
from these daemons to the application processes themselves. This is done by reliably mul-
ticasting the corresponding messages in the appropriate lightweight group; the lightweight
membership module at a daemon that receives such a message directs it to the correspond-
ing lightweight endpoint module. The lightweight endpoint module then takes appropriate
actions, and if necessary, passes the message to the application process as well.

Lastly, Section 4.4 details the actions between Starfish and users for employing manage-
ability.

**Dynamicy**

Starfish supports dynamic changes in the clusters. These changes can be the result of a node
being added or deleted from the cluster, or might happen due to failures and recoveries of
nodes. Each such change causes the group communication module to generate a new **view**
event, which reports this change to all members of the cluster. Also, changes that affect only
lightweight groups are reported in the lightweight group only, by the lightweight membership
module. One of the main benefits of using an underlying group communication system is
that our code does not need to explicitly keep track of such changes, since we can rely on
Ensemble to report these changes for us.

**High-Availability**

Starfish is a highly-available system, in the sense that a failure of a few nodes does not
cause the entire system to crash or hang. Instead, the system continues to run applications
and to be available to clients. In particular, if none of the application processes of a given
application was located on a failed node, then this application continues to run transparently.
Similarly, as discussed in the next subsection, even if the application had one of its processes
on a failed node, Starfish provides enough hooks to allow the application to overcome this
failure, either by restarting the failed process on a different node, or by restructuring the
computation.

Note that although the system as a whole does not stop due to a failure of a single node,
client connections to this node might be lost. However, if the client reconnects to the system,
he/she can continue the disrupted session from the point where the connection was cut off. At the moment clients have to explicitly choose the server they wish to connect to. However, in the future we plan to make this more transparent, using a one-IP type of solution [39].

4.3.2 Applications Dynamicity and Fault Tolerance

Dynamicity

Many applications can benefit from having more nodes added to them on-the-fly, while some applications might be able to accommodate the loss of a few nodes on-the-fly. This is typical in applications that have trivial parallelism, since in this case usually each node works independently on a given subset of the computation space. Thus, changing the number of nodes dynamically simply requires restructuring the computation subspace on which each node computes so that the entire compute space is covered with no duplicates.

Another feature of Starfish that allows applications to cope with dynamic changes is the C/R capability. Specifically, C/R allows Starfish to migrate application processes from one node to another, e.g., if a better node becomes available, or a new node is added to the cluster.

Fault Tolerance

Starfish offers two forms of fault-tolerance for applications: The main fault-tolerant mechanism employed by Starfish is C/R. The C/R module of Starfish is capable of performing any approach of distributed checkpointing at the application level, and performing both homogeneous and heterogeneous checkpointing at the process level, which is either system initiated or user initiated checkpointing. Thus, when a node failure occurs, Starfish can automatically restart the application from a checkpoint. This allows Starfish to run multiple C/R protocols side by side. In particular, we can compare various C/R protocols on the same platform.

The other form of fault-tolerance offered by Starfish is more application dependent, and is suitable mostly to applications that can be trivially parallelized. For such applications, whenever a node that runs one of the application processes crashes, a view event is delivered to all surviving application processes. This is done by having the application process register a listener handler with the object bus for membership events. (Note that applications that cannot utilize view changes simply do not register listeners for membership events, and their programming model remains the conventional MPI model.) Once the surviving members learn about the failure of a node, they can repartition the data sets on which each process computes, and continue to run without interruption.

When an application is submitted to Starfish, the client can also determine the fault-tolerant policy that should be applied to this application, i.e., whether automatic restart or view notifications should be used, and some rules regarding how to choose the node on which a process will be started after a partial failure. For compatibility, there is also an option to kill an application whenever one of the nodes dies in the middle of its execution, which mimics non fault tolerant systems.
4.4 The Starfish Control Center

The Starfish Control Center (for brevity, SCC) is a Java GUI program which allows for easy control, monitoring, and manipulating of Starfish [7]. This program enables the user to inspect, modify, and generally manage Starfish. SCC was written in Java, which is a platform-independent language, to keep Starfish portable and also let SCC as well as Starfish run on any architecture and OS.

As depicted in Figure 4.2, each Starfish daemon has a candidate endpoint for I/O connection. SCC initiates the connection to any Starfish daemon. Once SCC is started, the login dialog box, which is presented in Figure 4.4, will appear for determining the target node of Starfish and feeding security parameters. The Host field contains the IP address (or machine name) of any node of Starfish. The Port field contains the port number for TCP/IP connection with the Starfish daemon.

![Login to Starfish dialog box](image)

Figure 4.4: The login dialog box of SCC.

After the first login, the system will remember the parameters provided (except the password) and will use them as default values for future invocations. If the login fails, the program will start in a disconnected mode.

When logged into Starfish, SCC provides two different hierarchical views: By host and by application. User can switch views by clicking on the tabbed panes ”Host” and ”Applications”. Figure 4.5 presents a view of SCC, where users can monitor the running applications on Starfish.

Both views are presented as trees. As depicted in Figure 4.5, the ”Host” view shows a tree’s topmost of Starfish nodes. Under each Starfish node, the application nodes, and under each such application node, the rank nodes. Naturally, the application nodes under each host only show those applications that are currently running on the selected host, and the rank nodes under each application show only the ranks that belong to that application and are currently running on that host. On the other hand, the ”Application” view shows the application nodes in the tree’s topmost.

4.4.1 The Menu Commands

The Menu Commands contains almost every option available in every GUI menu. Below we describe in details some of the important options supported by the Menu Commands.
Starfish - This option contains the following commands for managing Starfish:

1. **Send Message**: This command is used for debugging purposes only. It opens a dialog box that allows to send any command message to the connected Starfish daemon.

2. **Logout**: Logs out of Starfish. The parallel program does not quit, though, and after logging out one can log-in again.

3. **Login**: This command is available only when SCC is not connected to any Starfish daemon. It opens the same login dialog box presented in Figure 4.4.

4. **Initialize**: Sends an initialization request to the Starfish daemon. Starfish performs some initialization before running any application.

5. **Shutdown**: Sends a shutdown request to the connected Starfish daemon. Note that the server may choose to ignore this command.

6. **Request Refresh**: Normally, Starfish notifies SCC upon any change. This command allows users perform refresh at any time. As a result, Starfish sends all the information for updating the window presented in Figure 4.5.

7. **Set environment Variable**: Each Starfish daemon maintains a list of environment variables. Users can use this command to change the value of any variable.
8. **Quit**: Attempts to log SCC out of the Starfish and unconditionally quits the program.

**Host** - This option contains commands for add/remove nodes. These commands can be performed only under administrator privileges.

1. **Add**: For adding a new host to Starfish.
2. **Remove**: For removing any host from Starfish.

**Application** - This option contains commands for application manipulations.

1. **Submit**: Opens the Submit dialog box for submitting (adding) a new application to Starfish. Figure 4.6 depicts the Submit dialog box in which users can submit an application with its parameters.

![Submit dialog box of SCC](image)

Figure 4.6: The submit dialog box of SCC.

In the **Application path** input box, the full pathname for the application’s binary executable should be provided. By default, Starfish will automatically select on which host(s) the new application will run. However, if users can explicitly select hosts by using the **Hosts** input field. Alternatively, you can press the **Select** button near the **Hosts** field to make a list-based selection. Pressing the button will bring up a host-selection dialog box, with a list of all currently available hosts. This dialog box contains some C/R fields for specifying the C/R policy and mechanism. The user should supply the checkpoint interval, the checkpoint file name, etc.

2. **Kill**: To kill any running application.
3. **Suspend**: To suspend any running application.
4. **Resume**: To resume any suspended application.
**Rank** - This option allows performing the following commands.

1. **Migrate**: To migrate a selected process to a different host.
2. **Request checkpoint**: Sends a checkpoint request to a selected process.

### 4.5 Experience Work

In this section we report some performance measurements conducted on Starfish. The performance measurements were obtained using 300 MHz Pentium II computers, connected by both Ethernet and Myrinet. In the case of Ethernet, we used the regular IP stack, while with Myrinet we used the BIP user-level interface [8].

#### 4.5.1 Checkpoint time

Figure 4.7 shows the homogeneous checkpointing time on Starfish using the sync-and-stop protocol [44]. Note that as expected, the checkpoint time grows linearly with the size of the checkpointed data. The time is on the order of seconds. Hence, if a checkpoint is taken once every hour, it would only slow down the entire execution time by less than 1%. Also, the hardware used for these measurements is not the most advanced, and employs regular IDE bus and controller. Newer and faster hardware is likely to result in faster saving times.

![Figure 4.7: Checkpointing in Starfish.](image)

Notice that in Figure 4.7, the smallest data point is 632KB, which takes 0.104 seconds for one node, 0.131898 seconds for two nodes, and 0.149219 for four nodes. This checkpoint size corresponds to checkpointing an empty program, or in other words, this indicates the checkpoint overhead imposed by our system. We attribute this low number to our architecture, in which the run-time system on each node is divided between the application process and the daemon. The daemon, which accounts for most of the code, is shared between all
processes on the same node, and is written in a way that we never have to save or recover its state. The only part of the run-time system that needs to be saved is the one that is included in the application process, and that part is relatively small.

4.5.2 Round-Trip Delay

In order to measure the application level round-trip latency, we have implemented a simple ping-style application. That is, one node sends a short message to another node, who immediately replies. We then measure the elapsed time between sending the message and receiving the reply at the application level. This is done repeatedly a hundred times to get the average round-trip latency. We have measured the round-trip delay with both TCP/IP and BIP/Myrinet. The results of these measurements are reported in Figure 4.8. It can be seen that the round-trip delay grows linearly with the size of the data.

![Figure 4.8: Round-trip delay vs. data size.](image)

The round-trip time for an empty message is 86 microseconds using BIP/Myrinet and 552 microseconds using TCP/IP, or in other words, roughly 43 microseconds one way with BIP/Myrinet and 226 microseconds one-way using TCP/IP. This time is the net overhead imposed by our system in handling a message, plus the network latency. It includes getting the message from the application and putting it on the network, and retrieving the message from the network, and then all the way back to the application. Also, these measurements were obtained on a non optimized prototype, running as bytecode. From our experience, the actual time for the optimized native code are expected to be much smaller.

4.5.3 System Overheads

Figure 4.9 reports on the time a message spends in each layer of our code. Here again, we refer to the non-optimized prototype, running as bytecode. Also, note that the time spent in each layer is independent of the message size, since messages are never copied in our code.
4.6 Using Starfish with NetSolve [17]

In this section we present a summary of a project that integrates three technologies, NetSolve [5], Starfish, and IBP [83], for the seamless integration of fault-tolerance into long-running applications. We discuss the design and benefits of this project.

NetSolve [5] is a brokered remote procedure call (RPC) environment as depicted in Figure 4.10. The user is termed a client, and is typically executing code on a PC or laptop. When the client wishes to perform a computationally complex task, he or she makes a NetSolve client call, specifying the name of the task, plus the arguments. The NetSolve client software manages the completion of this task, which we will refer to as a “service”.

First an agent is contacted with a query (step 1), specifying the service name and the size of the arguments. The agent maintains information on a collection of computational servers, which may be uniprocessors, multiprocessors, massively parallel machines, Condor workstation pools [68], etc. This information consists of machine parameters (speed, memory,
available software), plus current load information. The agent returns an ordered list of candidate servers to the client (step 2), who then picks a server (typically the first on the list) and initiates a RPC to that server (step 3). The server performs the service, and completes the RPC, returning the results to the client (step 4).

The Internet Backplane Protocol (IBP) \cite{83} is a mechanism for managing storage on the wide area. IBP servers are daemons that provide local storage (disk, tape, and physical memory) to remote clients that link the IBP client library. IBP is useful for checkpointing applications because it allows programs to store their checkpoints into a remote storage entity, perhaps one in a different administrative domain. Therefore if the machine executing the program fails and remains inoperative for a long period of time, the program may be restored on a separate machine, again perhaps in a different administrative domain.

### 4.6.1 Putting it all Together

The structure of NetSolve with checkpointing is depicted in Figure 4.11. In a nutshell, the NetSolve servers are linked with the checkpoint library of Starfish and store their checkpoints in IBP buffers. When a server fails, the computation is rolled back to the most recent checkpoint and restored on a new server. The client receives results from whichever server completes the computation. In such a way, the client ends up executing fault-tolerant and migratable code by simply linking with the NetSolve client library.

![Figure 4.11: NetSolve with checkpointing.](image)

### 4.6.2 Benefits of This Architecture

There are several benefits that this design has in terms of performance, functionality and deployability:

- **The user is insulated from checkpointing details.** In the best case, the user is employing NetSolve to perform common computations such as dense linear algebra. In this case, the NetSolve server setup is trivial, and the user can unknowingly receive the benefits of remote computation and checkpointing even while using Excel on a
Windows-based laptop. This is a level of deployability that is typically unheard of in scientific programming.

- **The user's program can have outside connections.** All checkpointing systems restrict connections outside the scope of the programming environment. In other words, while checkpointing systems typically work when all processors are part of the same programming system (for example through the use of PVM or MPI [36, 12]), they only allow programs to interact with the outside world by checkpointing (or logging) before each interaction. With NetSolve, the client may initiate a service while maintaining other external connections. This service can checkpoint, fail, rollback, and continue to operate correctly irrespective of the state of the client and its connections to other processing elements. This even works if the client starts the service asynchronously (i.e. in the background while it performs other tasks). Thus, NetSolve’s restricted programming model achieves a clean separation of client and server that allows the server to checkpoint while the client does other things.

- **Migration can occur across the wide area.** NetSolve and IBP both manage resources from different administrative domains, serving cycles and storage to potentially unrelated users and applications. With checkpointing to IBP, it is possible to migrate these services from one domain to another, so long as the server machine architectures are identical.

- **It will work in a limited-resource environment.** Similarly to the above, NetSolve and IBP are both able to manage spare resources (computation and storage) that have limits on their usage. In particular, processors may be revoked due to ownership, and storage may impose time limits on allocation. The inclusion of checkpointing into the NetSolve system means that these resources may be employed by remote computations. This functionality is similar to that provided by the Condor project [68].

- **Storage ownership is separated from the computation.** Pruyn and Livny have noted that strategic placement of checkpoints at locations external to the computation processors can improve performance [85]. The use of IBP in NetSolve is identical to the use of checkpointing servers in [85] and should improve performance similarly.

### 4.6.3 Performance Case Studies

We briefly detail three performance case studies. In each of these, we have a NetSolve client running Matlab, a NetSolve agent, two NetSolve servers and one IBP server all running on different machines. The Matlab client makes a NetSolve call to the `dmatmul` service (matrix multiplication), which gets serviced by one of the NetSolve servers. The server checkpoints to the IBP server, and either it completes without failure, or it fails. When the failure is detected, the second server takes over the service, reading from the checkpoint, and completes the service.

We report results from three separate computing environments: **Cluster**, **Local** and **Wide**. **Cluster** is a tightly-coupled cluster computing environment. The machines are all
dual-processor Sun UltraSPARC-2’s with 256 Mbytes of RAM, connected by a 155 Mbps ATM network. LOCAL is a department-wide environment, where the NetSolve client and agent are Sun UltraSPARC-1’s, and the other machines are lower-end SparcStation-5’s. All machines are connected by the Computer Science department’s backbone network at the University of Tennessee. Finally WIDE is a wide-area, multi-institutional environment where the client, agent and IBP server are running on UltraSPARC-1’s at Tennessee, while the NetSolve servers are running on two UltraSPARC-1’s at Princeton University. Communication between the two institutions is done over the standard Internet. In the CLUSTER test, the machines are dedicated to the experiment. In all other tests, the machines are undedicated.

Results from the CLUSTER environment are displayed in Figure 4.12. In this and other graphs, The light shaded areas are the server times only. The dark areas add the client interaction times. As expected, the CLUSTER environment exhibits high performance. The ATM network, large physical memories, and copy-on-write optimization combine for extremely high performance. For example, on the \( N = 1000 \) run, the overhead of checkpointing every ten seconds on the total client/server transaction is 9.7 percent, and the overhead of checkpointing every ten seconds and absorbing one failure is 26 percent.

Results from the LOCAL environment are displayed in Figure 4.13(a). As would be expected, the performance of the service is slower due to the slower processors. Likewise, the performance of checkpointing, recovery, and the contact with the client are all worse due to the slower interconnection network. However, in all cases, rolling back from the checkpoint improves performance over restarting from the beginning.

Finally, results from the WIDE environment are displayed in Figure 4.13(b). In these graphs, the black boxes are much larger due to the fact that the input and output matrices are being passed across the Internet. Interestingly, even though the checkpoints too are

Figure 4.12: Performance of \texttt{dmatmul} on the CLUSTER environment.
being passed across the Internet, the checkpoint overhead is negligible in comparison to the fluctuation due to non-dedicated access. Once again, this is due to the copy-on-write optimization. However, when a recovery is required, the checkpoint file must be moved across the Internet before recovery may begin, resulting in a severe performance penalty. In this instance, a restart from the beginning would perform better than restarting from the checkpoint. This experiment serves to underscore that it is more important to select the recovering server to be close to the checkpoints than it is to select the checkpointing server to be close to the checkpoints. This is because checkpoints are taken asynchronously, while state restoration is by nature synchronous.

![Figure 4.13: Performance of dmatmul on the LOCAL and WIDE environments.](image)

(a)  
(b)

Technion - Computer Science Department - Ph.D. Thesis PHD-2002-01 - 2002
Chapter 5

Heterogeneous Checkpointing

5.1 Overview

This chapter reports on our experience in constructing a heterogeneous C/R mechanism for Starfish using the OCaml virtual machine [10]. This mechanism operates across a wide range of Unix-like operating systems as well as Windows NT, and across multiple hardware platforms such as Intel’s Pentium, Compaq’s Alpha, Sun’s UltraSparc, and IBM’s RS/6000. Additional ports to any other platform should be fairly trivial, given that we already support both little-endian and big-endian formats, and both 32-bit and 64-bit architectures. Our implementation relies on the structure of the OCaml virtual machine, and the fact that it is the same on all these platforms. We utilize the garbage collection features of OCaml and the tagging of data types in dumping and reconstructing the data across platforms. We also employ the natural optimization that data translation between representations is done only during recovery time, and only if recovery occurs on a different architecture than the one in which the application was saved.

The rest of this chapter is organized as follows: Section 5.2 includes a description of OCaml, its virtual machine and garbage collection mechanism. We review the main problems in trying to implement heterogeneous C/R and our implementation in Section 5.3. A qualitative analysis and performance measurements are included in Section 5.4. Lastly, in Section 5.5 we describe the integration of the heterogeneous checkpointing with Starfish.

5.2 The OCaml Virtual Machine

OCaml [10] is a dialect of the ML programming language. The OCaml system supports both native code and bytecode compilation. For native code, the compiler generates an optimized machine-dependent executable code. On the other hand, the bytecode compiler generates hardware-independent code that can run in any machine architecture or OS for which there is an OCaml runtime environment.

The OCaml Virtual Machine (OCVM) is implemented in C with well documented source code. It is also open source, which allows users to add or modify the system. Moreover,
OCVM is ported to most platforms and operating systems. All this makes OCVM a good vehicle for investigating heterogeneous C/R at the virtual machine level.

In this chapter we describe the low level design and implementation of OCVM. This description is useful for understanding the issues involved in implementing heterogeneous C/R. We first describe data representation and manipulation. We then describe the main memory components of the system: the heap, the stack, and the bytecode, followed by a brief description of how threads are created and managed by the system. Lastly, we detail the garbage collector and the interpreter mechanisms of the system. The purpose of this description is to obtain better understanding of OCVM and thereafter the C/R mechanism that we integrated with OCVM.

5.2.1 Main Memory Areas

OCVM has four main memory areas: the heap, the stack, the bytecode, and a set of three registers. These areas are allocated by OCVM during initialization. We explain below these components and their purpose in the system.

Heap The main purpose of the heap is to maintain application data. OCVM allocates both static and dynamic data in the heap. Memory deallocations in the heap are performed implicitly by the garbage collector (see Section 5.2.4 for more details).

Initially, OCVM allocates a heap of a predefined size. The heap is expanded if necessary during runtime according to applications’ requirements. OCVM maintains the heap as a linked list of memory chunks ordered by increasing memory addresses. Each chunk consists of a chunk header, which holds its length and a pointer to the next chunk. Initially, all the heap space is declared free and added to the freelist data structure. During execution, allocations are done by allocating objects from freelist, and a deallocation returns an object to freelist.

Stack During initialization, the stack is allocated using malloc with a default size of 16K. If the stack becomes full, OCVM reallocates a new stack with double the size of the old one. The main purpose of the stack is, as usual, to save local variables and return addresses of function calls. However, local variables of non-primitive data types are actually dynamically allocated in the heap, and the stack only includes a pointer to them.

Bytecode During initialization, OCVM allocates an area of memory by using malloc for maintaining the OCaml program bytecode. It is generated by using the bytecode compiler and the same file can be loaded and executed in any architecture type.

Registers OCVM defines three virtual registers for executing bytecode:

PC: The program counter register points into the bytecode memory where the next instruction will be fetched.
SP: The stack pointer register points into the stack head.
ACCU: The accumulator helps in maintaining temporary values for the interpreter.
5.2.2 Data Representation

The basic data type of OCaml is a 32-bit word in a 32-bit architecture and a 64-bit word in a 64-bit architecture. In our description below we assume a 32-bit data representation. The discussion for a 64-bit architecture is analogous. Usually, this basic data type contains a value of either an integer or a pointer. For an integer, the least significant bit is always set to 1. A pointer value in OCaml is a 4-byte aligned address that points into one of the main memory areas.

OCaml uses two kinds of data types: atomic and blocked. An atomic data type is a 32 bits in size, which is either a small integer value or just a 0-tuple block. Atomic types of 0-tuple blocks are allocated statically once and for all the program’s life outside the heap. These types are used by OCaml to represent a user-defined abstract type. On the other hand, blocked types are used to represent more complex data structures that are allocated in the heap. For each blocked data there is a strictly defined 32-bit header consisting of three fields, namely tag, color, and size. The first 8 bits of the header contains the block tag, which specifies the block type among various blocked types in OCaml. The next two bits of the header contain the block color, or state, in the garbage collection mechanism. The last 22-bit field contains the block size, which is the number of the block fields except for the header. The other entries in a block data are also 32 bits each. Moreover, each of those fields is simply an OCaml value that contains either an actual value or a pointer value. Figure 5.1 shows an example of a blocked data type with four fields.

Figure 5.1: A blocked type with its header as defined by OCaml.

5.2.3 Threads

From the operating system’s point of view, OCVM is a regular process. Thus, OCVM has its private data, text, and stack memory segments allocated by the operating system. Consequently, OCVM defines and creates its own stack, heap, registers, etc., in which it provides a runtime environment and executes the bytecode application.

OCVM supports both single and multi-threaded applications. In the case of a multi-threaded application, OCVM executes and manages all the threads using its own user-level thread package. Each thread has a private stack allocated for it by OCVM. OCVM schedules
5.2.4 Garbage Collection

A Garbage Collection (GC) algorithm periodically scans the heap in order to reclaim unreachable objects. In OCaml, garbage-collectible data are blocked types allocated in the heap, as described before. The garbage collector recognizes a garbage-collectible block by using the tag value which is defined in each block header and presented in Figure 5.1.

OCaml uses an algorithm for garbage collection that combines the generation and mark-sweep algorithms [40, 61]. Specifically, dynamically allocated data is held by OCaml in two areas, the young generation and the old generation areas [41, 42]. The young generation is a fixed area of memory defined between the young_start and young_end variables, which is where values are initially allocated. A minor garbage collection is a collection of the young generation, in which all reachable values are copied from the young generation into the old generation. The space used for the young generation is recycled after a minor garbage collection.

On the other hand, the old generation area is simply the heap, which is a linked list of memory chunks ordered by increasing memory addresses. Chunks are added to the old generation by calling malloc as necessary, and a page table is used to distinguish memory pages belonging to a chunk in the old generation from other memory. A major garbage collection works on the old generation. This operation uses the incremental mark-sweep algorithm, in a number of slices. One slice of a major garbage collection is executed after every minor garbage collection operation. There are two kinds of garbage collection slices: mark slices and sweep slices. A sequence of mark slices (called the mark phase) followed by a sequence of sweep slices (the sweep phase) constitute one cycle of a major garbage collection. The amount of marking (resp. sweeping) to do in a mark (resp. sweep) slice is determined by the total size of the reachable values being promoted from the young generation in the preceding minor collection: the more promotions, the more garbage collection work must be done. This is an attempt to distribute the garbage collection work over computation in a fair manner, ideally ensuring that the application never has to wait long for the collector to free memory.

5.3 Heterogeneous Checkpoint and Restart

Figure 5.2 shows the internal structure of OCVM where we added the C/R module for implementing heterogeneous C/R [16]. By implementing the C/R inside the virtual machine, we have full control over the OCVM scheduler and its main memory areas, which simplifies our task.

In this section we present some issues that are significant for implementing heterogeneous C/R at the virtual machine level. These issues are common to any such implementation,
5.3.1 Checkpoint Issues

Performance

In order to avoid blocking the application during checkpoint, it is possible to fork a new process. The child process then saves its state and exits. In saving the data, there is a tradeoff between the overall size of the saved file and the number of write operations. The easiest thing is to simply dump everything to disk. On the other hand, in order to minimize the size of the checkpoint file, more sophisticated mechanisms must be implemented, as described, e.g., in [81]. Such mechanisms, however, require scanning parts of the memory, and involve more write operations to the disk, thereby creating additional overhead.

Safe Points

Each bytecode instruction is interpreted by the virtual machine into several native machine instructions. In particular, the execution of each bytecode instruction by the OCVM interpreter involves reading the next instruction, then decoding it, and finally executing it. This is analogous to the way hardware executes a (native) machine instruction.

A system initiated checkpoint operation might be invoked in arbitrary moments during execution. Thus, it is possible that a checkpoint will occur in the middle of executing a single bytecode instruction, which might result in saving an inconsistent state of the application. That is, since we are operating in the virtual machine, using bytecode granularity, during restart, the system will either have to reissue the instruction, or start from the next instruction. However, reissuing the instruction might not be legal, if the part that was executed is not idempotent. On the other hand, skipping the instruction at restart is also not desirable, since the instruction did not terminate, e.g., did not update the memory or registers, etc. Hence, special care should be taken to avoid saving the state of the virtual machine in the middle of a bytecode instruction.
We define a *safe point* to be a point in the execution in which a checkpoint would save a consistent application’s state. A safe point in a bytecode application can be found between every pair of subsequent instructions or during an instruction that does not change the system’s state. In our approach we consider safe points. When a checkpoint is invoked, we set a particular flag indicating a checkpoint request and continue normal execution. Note that the OCVM interpreter executes bytecode instructions one at a time. Between the completion of one instruction and the beginning of the next one, the interpreter checks the flag, and if necessary, initiates a checkpoint before the new instruction starts executing.

**Checkpointed Data**

The *checkpointed data* is the data that has to be saved during checkpointing. In a homogeneous checkpointing mechanism, the checkpointed data simply consists of the process data segment, heap, stack, and its registers. In heterogeneous checkpointing, however, the checkpointed data has to be defined in the virtual machine equivalents of the above. That is, we do not save the virtual machine process, but rather save the logical state of the application as maintained by the virtual machine.

**Thread Synchronization**

There are several points that should be taken into account when developing a checkpoint mechanism for multi-threaded applications. First, every checkpoint $C$ in a multi-threaded application should be *consistent*. That is, every read from a global data that occurs before $C$ must read a value that is written before $C$. An easy way to ensure consistent checkpointing for multi-threaded applications is to stop all threads, take the checkpoint, and then resume normal multi-threaded execution.

Another problematic point of checkpointing a multi-threaded application is obtaining the status of all live threads in the application. OCVM implements its own user-level threads, and thus this information is accessible inside the virtual machine. Otherwise, one has to rely on kernel support to obtain this information. Even worse, the information regarding OS native threads might not be portable, and some translation mechanism might be needed in these cases.

**5.3.2 The Checkpoint Mechanism**

As discussed before, in order to avoid taking a checkpoint in an unsafe point, whenever it is time to take checkpoint, we set a checkpoint flag. Figure 5.3 depicts a pseudo code of the checkpoint mechanism. This code helps understanding the following steps:

1. If the application is multi-threaded, disable thread scheduling and flush all file channels. Then create a new process for performing copy-on-write checkpointing [17, 81]. The child process performs the checkpoint while the parent process re-enables thread scheduling and continues normal execution.
2. Invoke a minor garbage collection to reclaim garbage in the young generation. This way, there is no need to save the young generation.

3. Open a temporary checkpoint file for saving the checkpointed data. Then save the actual machine information such as the architecture type (32-bit or 64-bit and big-endian/little-endian).

4. Save relevant boundary addresses of main memory areas. Then save the checkpointed data as mentioned in Section 5.3.1.

5. Save information about the opened channels. Channels maintain abstract information about the opened files and this information is important for restoring file descriptors.

6. Write a signature at the end of the temporary checkpoint file. Flush and close the file. Then, after completing the checkpoint successfully, set the temporary file to be the new checkpoint file. The use of a temporary file while saving the data avoids losing the checkpoint file if a failure occurs during checkpointing.

```c
int checkpoint(value sp, code_t pc, value accu, value env, long extra_args)
{
    int rc, chkptFd;
    ...
    gc_minor(Val_unit);
    pid = fork();
    if (pid >0) /* Parent process */
        exit(0);
    if (MULTI_THREADED)
        disableTimer();
    chkptFd = open(tempChkptFileName, O66, 022);
    saveBoundaryAddresses(heap_start, start_code, stack_high);
    saveArchtCharacters();
    CHKPT_WRITE(chkptFd, atom_table, wsize); /* Save atomics */
    rc = saveFreeList(chkptFd);
    /* Save OCVM globals */
    globals.global_data = global_data;
    globals.allocated_words = allocated_words;
    globals.extern_sp = extern_sp;
    CHKPT_WRITE(chkptFd, &globals, sizeof(globals));
    /* Save the OCaml registers */
    registers.sp = sp;
    registers.pc = pc;
    registers.accu = accu;
    registers.env = env;
    registers.extra_args = extra_args;
    CHKPT_WRITE(chkptFd, &registers, sizeof(registers));
    dumpHeap(chkptFd, heap_start);
    saveStack(stack_start, sp);
    if (MULTI_THREADED)
        saveThreadInfo();
    saveChannels();
    closechkptFd;
    link(tempChkptFileName, chkptFileName);
    unlink(tempChkptFileName);
    exit(0);
}
```

Figure 5.3: Relevant source code of the checkpoint function in OCVM.
5.3.3 Restart Issues

Data Representation

Nowadays, data representations typically differ in two aspects: big-endian vs. little-endian convention and 32-bit vs. 64-bit word size. To make our work general, we should be able to restart an application on any architecture type regardless of the original architecture.

During restart, we adjust numbers representation to the new architecture. For atomic types and pointers, we convert each datum according to the target machine. On the other hand, we keep OCaml blocked types unchanged. For instance, floating point in OCaml is a blocked type allocated in the heap, as discussed in Section 5.2.

We assume that application failures are rare, and therefore, the overhead for checkpointing should be minimal. Thus, we prefer to save data in its native representation. During restart, data is restored according to the machine on which it is being restarted.

Pointers Adjustment and Values Distinction

Upon restart, we restore all the checkpointed data and allocate them in memory areas as they were in the original checkpointed process. However, we cannot guarantee that all data will be allocated in the same addresses as in the original machine. Hence, we need to adjust each pointer to point to the relevant new address.

In order to properly adjust pointers in the restarted application, we first need to differentiate pointers from non-pointer values. Then, for each pointer, we need to adjust its contents to point to the corresponding cell in the new machine. In OCVM we can differentiate pointer from non-pointers by examining the least significant bit as mentioned in Section 5.2.2.

During checkpointing, we save the memory boundaries of all these areas. During restart, using these boundaries we examine for each value if it is a pointer and to which memory area it was pointing. Then, we adjust the pointer to the new address by adding the offset to the beginning of the specified memory area.

Thread Synchronization

Thread synchronization is also required during restart. Note that allowing some threads to start executing before all threads are recreated and the program completely restored may result in errors and/or deadlocks. These problems can be solved, but at the cost of increased complexity. Thus, for the sake of simplicity, we chose to prevent all threads from executing until the entire program, including all its threads, has been restored.

File Descriptors

In general, restoring file descriptors is one of the traditional restrictions of C/R [68, 69]. Particularly, in the heterogeneous C/R case this problem becomes even more difficult because file systems may differ between two different machines and/or OSs. Thus, we prefer not to deal with this problem individually for each system. Instead, we need to tackle this problem within the virtual machine and use an abstract mechanism for heterogeneous C/R.
In our case, OCVM allocates a particular structure called *channel* for each opened file descriptor. A channel maintains relevant information about the corresponding opened file. In particular, when writing data to an open file, the data is written to a buffer maintained within the channel data structure. When the channel is flushed, this buffer is also flushed to disk. Thus, during checkpoint, we first flush the channels’ buffers to disk, and then we save their state. During restart, we can reopen all corresponding files and seek to their position at checkpoint time.

### 5.3.4 The Restart Mechanism

We have modified OCVM such that during initialization, OCVM accepts as a parameter whether the application should be started from the beginning or from a given checkpoint file. Figure 5.4 depicts the relevant code of the restart mechanism, as presented below step-by-step.

1. Open the provided checkpoint file and check if it has a valid signature.
2. Read the original machine information. Determine the original machine big-endian/little-endian representation and its architecture bit size (32-bit or 64-bit). Finally, determine the application type as either multi-threaded or single-threaded.
3. Read the original boundary addresses for adjusting pointers according to their values in the original machine. Then restore the abstract registers.
4. Restore the heap and the stack. First, expand their memory area if needed and then overwrite the original data from the checkpoint file. Finally, adjust the pointers. In the heap, use the garbage collection mechanism to visit only the reachable and live blocks, then inside each block fix the pointers accordingly.
5. Restore channels and adjust pointers there. For each channel, open its corresponding file and then restore its information.
6. Reenable thread scheduling if needed.

### 5.4 Evaluating Heterogenous C/R

#### 5.4.1 Advantages, Disadvantages, and Restrictions

The most important advantage of this C/R mechanism is the ability to do cross platform checkpoint and restart. Yet another advantage that stems from implementing C/R inside the virtual machine is the ability to overcome some of the typical restrictions of user space C/R. This is because the virtual machine manages and operates the application in a similar manner to the kernel of an OS. Since we only dump the checkpointed data of the virtual machine, the overall size of the checkpoint file is smaller than in implementations that dump the
int restart(value **sp, code_t *pc, value *accu, value *env, long *extra_args)

int rc, chkptFd;
....
chkptFd = open(chkptFileName, O_RDONLY, 0);
restoreArchtChars(chkptFd);
/* Read original address boundaries */
CHKPT_READ(chkptFd, baseAddr, sizeof(baseAddr));
CHKPT_READ(chkptFd, registers, sizeof(registers));
fixPointers(registers, baseAddr);
restoreHeap(chkptFd, heap_start);
CHKPT_READ(chkptFd, atom_table, wsize);
rc = restoreFreeList(chkptFd);
CHKPT_READ(chkptFd, globals, sizeof(globals));
fixPointers(globals, baseAddr);
CHKPT_READ(chkptFd, oldStack_sz, sizeof(int));
if (oldStack_sz > stack_sz)
realloc_stack(oldStack_sz);
restoreStack(chkptFd, sp, stack_high);
fixStackPointers(sp, baseAddr);
/* Fix pointers in the heap */
chunk = heap_start;
while (chunk != NULL) {
  p = (header_t *)chunk;
  chend = chunk + Chunk_size (chunk);
  while ((void *)p < chend){
    hd = Hd_hp(p); sz = Wosize_hd(hd); tg = Tag_hd(hd);
    if ((!Is_blue_hd (hd)) && ((tg < No_scan_tag) || (tg == Final_tag))) {
      vp = p + 1;
      for (i = 0; i < sz; i++, ++vp)
        if (Is_block(*vp)) fixPointer(*vp, vp, baseAddr);
      p += Whsize_wosize (sz);
    }
    chunk = Chunk_next (chunk);
  }
restoreChannels(chkptFd);
}

Figure 5.4: Relevant source code of the restart function in OCVM.

entire core. Another advantage of this mechanism is transparency. Unlike other techniques
of heterogeneous C/R [45, 67], our mechanism does not require any modification in either
the user code or during compilation and linking [82].

On the other hand, our C/R mechanism is implemented in a virtual machine that executes
bytecode. Bytecode usually executes slower than native code, due to the need to interpret
it in software.\(^1\) Hence, an interesting research topic, which we have started to investigate,
is extending our work to native code OCaml.

Our work still has a few restrictions, which are typical in works that do not affect the
application’s programming model. These include:

- Outside connections, like sockets, are not supported. If a failed application has an
  outside connection, our C/R mechanism cannot recover this connection and the
  application might not run properly. Note that sockets are problematic, since the other
  end might not implement any kind of fault tolerance mechanism, in which case it is
  meaningless to try to reopen the socket on the computer that performed the checkpoint.

- We can recover file descriptors, but only if the same file is accessible from the restarting
  machine. Also, files can only be read from, or written to, sequentially. When a file
  is written sequentially, during restart we simply seek the file to the position it had

\(^1\)See http://caml.inria.fr/ocaml/speed.html for a discussion about the difference in speed between byte-
code and native code OCaml.
Table 5.1: A list of different machines that have been tested with heterogeneous C/R

<table>
<thead>
<tr>
<th>Machine Name</th>
<th>Architecture Type</th>
<th>OS</th>
<th>Representation</th>
<th>Word Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>rodrigo.cs.technion.ac.il</td>
<td>Intel P-II 350MHz, i686</td>
<td>RedHat 6.1 Linux</td>
<td>little-endian</td>
<td>32-bit</td>
</tr>
<tr>
<td>csd.cs.technion.ac.il</td>
<td>Sun Ultra Enterprise 3000</td>
<td>SunOS 5.7</td>
<td>big-endian</td>
<td>32-bit</td>
</tr>
<tr>
<td>mathcs11.haifa.ac.il</td>
<td>RS/6000</td>
<td>AIX 3.2</td>
<td>big-endian</td>
<td>32-bit</td>
</tr>
<tr>
<td>dock.cs.technion.ac.il</td>
<td>Intel P-I, 160MHz</td>
<td>FreeBSD 3.2</td>
<td>little-endian</td>
<td>32-bit</td>
</tr>
<tr>
<td>pc8.cs.technion.ac.il</td>
<td>Intel P-II, 350MHz</td>
<td>Win NT</td>
<td>little-endian</td>
<td>32-bit</td>
</tr>
<tr>
<td>spe148.testdrive.compaq.com</td>
<td>Dual Alpha DS20 500MHz</td>
<td>RedHat 6.2 Linux</td>
<td>little-endian</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

during restart. However, if the file is written randomly, we cannot guarantee correct semantics, unless we also implement a log file, which we have not so far.

- C/R restricts interleaving of C and OCaml. That is, since we only checkpoint memory allocated inside the virtual machine’s heap, any C part of an OCaml application must only use the OCaml library routines for allocating and deallocating memory, rather than using the OS native ones.

5.4.2 Experimental Results

We tested our implementation of heterogeneous C/R on several hardware architectures and OSs, as listed in Table 5.1. To verify heterogeneous C/R, we have performed C/R across these distinct platforms. As can be seen, this represents a wide selection of different platforms on which heterogeneous C/R works.

C/R Overhead

In this section we present some performance measurements done on the heterogeneous C/R mechanism that we built for Starfish. As shown below, the checkpoint overhead is proportional to the size of the application data. There are two reasons for this: On all Unix-like systems, checkpoint is taken in a forked process. On systems that implement copy-on-write fork, each write causes an exception, which adds an overhead, so the more data there is, the more exceptions will occur. On the other hand, in systems that fully duplicate the memory during fork, this duplication takes time that is linear in the memory size. The other reason is that we save the state on stable storage. Thus, even if this is done on a separate process, it consumes systems resources, and slows down the original application process.

We have two test applications for C/R measurements, namely matrix multiplication and Fast Fourier Transform (FFT) [76]. In both cases, checkpointing is invoked periodically by the system.

Figure 5.5 shows the checkpoint overhead, when running on rodrigo (a Linux machine). The X axis represents the checkpoint size in MBs, and the Y axis represents the runtime in seconds. We can see that for matrix multiplication the checkpoint overhead is almost negligible. This is because only two thirds of the data are read-only, and since we use copy-on-write checkpointing mechanism, the checkpoint overhead is affected only by one third of
the checkpointed data. On the other hand, in FFT, none of the data is read-only, and thus the gap between the runtime with and without checkpoint is much more noticeable.

Figure 5.5: Comparing the running time of matrix multiplication and FFT respectively with and without checkpointing on rodrigo.

Figure 5.6 shows the restart overhead of the matrix multiplication test program on the various platforms mentioned in Table 5.1, when the checkpoint file was created on rodrigo. Notice that since rodrigo and PC8 are of the same architecture, the restart overhead on these machines is almost equal. However, the increase in restart overhead for csd is caused by the need to convert each value from little-endian to big-endian. In sp2148, the difference is attributed to the need to expand each value from 32 bits to 64 bits.

Figure 5.6: Comparing the restart time on different platforms.
Timing for Substantial Parts of the C/R Mechanism

In this section we measure the checkpoint latency. Particularly, we measure the time spent in the substantial parts of this mechanism in the checkpointer process during checkpoint. By substantial we mean those parts that might have noticeable contribution to the overall checkpoint latency. For these measurements, we used the matrix multiplication test on rodrigo running Linux with the ext2 file system. Note that due to the use of fork, these times do not directly affect the checkpoint overhead, which is measured in comparison to the original process running time.

There are two main substantial parts in checkpointing: saving the heap area in stable storage and committing the checkpoint file. We notice in Figure 5.7 that more than 80 percent of the checkpoint time is spent in saving the heap. The bigger the checkpoint file becomes, so does the time for committing it. The time to commit mainly comes from deleting the old checkpoint file. In order to delete large files on ext2, it may be necessary to read all associated inodes, which could span several hierarchies, in order to find and free all blocks belonging to the file.

We notice also in Figure 5.7 that other parts take less than 5 percent of checkpointing. These parts contain the minor garbage collection, saving the abstract registers, and saving the stack. All these operations are performed either in a small area of memory or on a few distinct values.

During restart, the substantial parts are restoring the heap and fixing pointer values inside it. Figure 5.7 depicts the restart time in seconds. We notice here that these substantial parts take more than 90 percent of restart. Note that there are another two substantial parts of restart that we have already evaluated in the previous section. The first part is the time needed for converting from big-endian to little-endian representation or vice versa. The second part is the time needed for converting from 32-bit to 64-bit word size representation or vice versa. Figure 5.6 exhibits those times.

![Figure 5.7](image-url)

Figure 5.7: Timing the substantial parts on rodrigo vs. various checkpointed data size.
5.5 Heterogeneous C/R in Starfish

As we mentioned before, due to heterogeneous C/R, Starfish is considered as a reliable heterogeneous distributed system. However, in order to provide heterogeneous C/R in Starfish as proper, Starfish should run on the modified OCaml virtual machine with the heterogeneous C/R module as depicted in Figure 5.2. Notice here that the C/R module in Starfish needs to apply distributed C/R protocol, where the heterogeneous C/R module of Figure 5.2 is considered the Checkpointer as depicted in Figure 3.1 (see Chapter 3).

As a result, Starfish now supports both homogeneous and heterogeneous checkpointing mechanisms for local checkpointing. The user is expected to compile his or her application with either the modified OCVM for supporting heterogeneous checkpointing or with the user library of checkpointing in Starfish as described in Appendix A.

Figure 5.8: Comparing local homogeneous and heterogeneous checkpointing in Starfish.

Figure 5.8 depicts a comparison between the checkpoint overhead of the homogeneous (in Figure 5.8(a)) and heterogeneous (in Figure 5.8(b)) mechanisms. These figures show the checkpoint overhead in Starfish by using the same distributed checkpointing protocol: sync-and-stop.

Heterogeneous checkpointing time is depicted in Figure 5.8(b), where the smallest data point is 260KB, which takes 0.0077 seconds for one node, 0.0205 seconds for two nodes, and 0.052 seconds for four nodes. In this case the checkpointed data does not contain the virtual machine data which is saved in the homogeneous case.
Chapter 6

Distributed Checkpointing

6.1 Overview

As a reliable high performance distributed system, Starfish supports distributed checkpointing protocols for providing fault-tolerance. In this chapter, we address the C/R issues that are particular to distributed settings, providing efficient C/R protocols and guaranteeing recovery lines.

As discussed in Chapter 1, there is an important tradeoff between the overhead imposed by the protocol to guarantee a recovery line and the number of checkpoints that must be rolled back after a failure. The protocol overhead can be in the form of message and memory overhead, as well as in terms of forcing processes to take checkpoints more often than they would otherwise prefer. One extreme approach to this tradeoff is coordinated checkpointing. The two common techniques for coordinated checkpointing are either to synchronize and pause the execution of all processes until everyone has taken a checkpoint [44, 81], or to use Chandy-Lamport’s distributed snapshot protocol [34]. The other extreme approach is uncoordinated checkpointing. This, of course, does not impose any overhead, but runs the risk of never generating a recovery line [44, 104].

In this chapter we present a classification of distributed executions based on the maximum number of checkpoints that may need to be rolled back after a failure in order to recover from a recovery line. Then we present a distributed checkpointing protocol that guarantees a $d$-BC execution.

6.2 Definitions and Notations

We start with a definition of a zigzag path as introduced by Netzer and Xu [75].

**Definition 6.2.1:** A zigzag path (also called Z-path) from $C_{p,i}$ to $C_{q,j}$ (denoted as $C_{p,i} \rightarrow C_{q,j}$) is a sequence of messages $(m_1, m_2, \ldots, m_l)$, $l \geq 1$, such that

1. $m_1$ is sent by process $p$ after $C_{p,i}$,
2. if \( m_k (1 \leq k < l) \) is received by process \( r \), then \( m_{k+1} \) is sent by \( r \) in the same or later checkpoint interval (\( m_{k+1} \) may be sent before \( m_k \) is received), and

3. \( m_l \) is received by process \( q \) before \( C_{q,j} \).

In particular, a Z-path can be from a checkpoint to itself, in which case, it is called a Z-cycle, and the checkpoint is said to be in a Z-cycle.

A process \( p \) is a participant in a Z-path \( \xi \), denoted \( p \in \xi \), if it sends or receives a message in \( \xi \). The weight of \( \xi \), denoted \( \omega(\xi) \), is the number of participants in \( \xi \). Below we define the pre-border and post-border of a Z-path which are the checkpoints before and after the Z-path, respectively.

**Definition 6.2.2:** For each participant \( p \) in a Z-path \( \xi \), the pre-border of \( \xi \) on \( p \), denoted \( \text{pre}_p(\xi) \), is the checkpoint \( C_{p,k} \) such that the first event of either sending or receiving a message in \( \xi \) by \( p \) belongs to \( I_{p,k+1} \). The pre-border of \( \xi \), denoted \( \text{pre}(\xi) \), is the set \( \{\text{pre}_p(\xi) | p \in \xi\} \).

Similarly, the post-border of \( \xi \) on \( p \), denoted \( \text{post}_p(\xi) \), is the checkpoint \( C_{p,k} \) such that the last event of either sending or receiving a message in \( \xi \) by \( p \) belongs to \( I_{p,k} \). The post-border of \( \xi \), denoted \( \text{post}(\xi) \), is the set \( \{\text{post}_p(\xi) | p \in \xi\} \). A checkpoint between \( \text{pre}_p(\xi) \) and \( \text{post}_p(\xi) \) is an internal checkpoint of \( \xi \).

![Figure 6.1: An execution example](image)

Figure 6.1: An execution example: \( \xi_1 = (m_4, m_5) \) is a Z-path from \( C_{1,1} \) to \( C_{3,2} \). \( \xi_2 = (m_7, m_4) \) is a Z-cycle from \( C_{2,2} \) to itself, where \( P_1, P_2 \in \xi_2 \).

In Figure 6.1, \( \{C_{1,1}, C_{2,1}, C_{3,1}\} \) and \( \{C_{1,2}, C_{2,3}, C_{3,3}\} \) are the pre- and post-borders of the Z-path \( (m_4, m_5, m_6) \); \( \{C_{1,1}, C_{2,1}\} \) and \( \{C_{1,2}, C_{2,3}\} \) are pre- and post-border of the Z-cycle \( (m_7, m_4) \).

The notion of a cycle extends the notion of a Z-cycle and lies at the heart of our discussion on rollback propagation (presented in Chapter 6).

**Definition 6.2.3:** A Z-path from \( C_{p,i} \) to \( C_{q,j} \) is a cycle if \( p = q \) and \( j \leq i + 1 \). Namely, a cycle is a Z-path from a checkpoint to either an earlier checkpoint on the same process, or itself, or the next checkpoint on the same process.

The tail of a cycle \( \Phi \), denoted \( \text{tail}(\Phi) \), is all checkpoints \( C \) such that \( \text{pre}(\Phi) \sim C \). In Figure 6.1, \( \Phi = (m_1, m_9) \) is a cycle but not a Z-cycle. Observe that \( \text{pre}(\Phi) = \{C_{1,0}, C_{2,0}\} \) and \( \text{post}(\Phi) = \{C_{1,1}, C_{2,1}\} \). On the other hand, the cycle \( \Phi' = (m_4, m_7) \) is a Z-cycle from \( C_{2,2} \) to itself. Moreover, \( C_{3,1} \in \text{tail}(\Phi) \) since \( m_3 \) constitutes a Z-path from \( \text{pre}(\Phi) \) to \( C_{3,1} \).
Claim 6.2.1:  Given a cycle $\Phi = (m_1, m_2, \ldots, m_k)$, if $p, q \in \Phi$, $C_{p,i} \xrightarrow{hb} post_p(\Phi)$ and $pre_q(\Phi) \xrightarrow{hb} C_{q,j}$, then there is a Z-path from $C_{p,i}$ to $C_{q,j}$.

Proof: A rotation of a sequence of messages $(m_1, \ldots, m_n)$ by $k$ messages is the sequence $(m_{n-k}, \ldots, m_1, \ldots, m_{n-k-1})$. Clearly, the rotation operation can be applied to cycles. It is straightforward to see that a rotation of a cycle is a cycle with the same participants and borders.

By definition, there is a message $m \in \Phi$ that was sent by $p$ after $C_{p,i}$ and there is another message $m' \in \Phi$ that was received by $q$ before $C_{q,j}$. Consider the rotation $(m, \ldots, m', \ldots)$ of $\Phi$ starting with $m$. Clearly, its prefix $(m, \ldots, m')$ is a Z-path from $C_{p,i}$ to $C_{q,j}$. $\square$

Corollary 6.2.2: For each participant $p$ of a cycle $\Phi$, $pre_p(\Phi) \xrightarrow{\sim} post_p(\Phi)$. Moreover, every internal checkpoint of $\Phi$ is in a Z-cycle.

Given a cut of checkpoints $S$, $P(S)$ is the set of processes that have checkpoints in $S$. For $p \in P(S)$, $S|_p$ denotes the checkpoint of $p$ in $S$. We say the cut $S$ is full if it contains a checkpoint from every process, otherwise, it is partial.

The notion of a Z-path which we defined for a pair of checkpoints is also applicable to cuts: Given a checkpoint $C_{p,i} \not\in S$, we denote by $C_{p,i} \xrightarrow{\sim} S$ (or $S \xrightarrow{\sim} C_{p,i}$) the fact that there is a Z-path from $C_{p,i}$ to $S|_q$ (or from $S|_q$ to $C_{p,i}$) for some process $q \in P(S)$. Furthermore, given two cuts $S_1$ and $S_2$, $S_1 \xrightarrow{\sim} S_2$ denotes the existence of a Z-path from some checkpoint in $S_1$ to some checkpoint in $S_2$.

Definition 6.2.4: A cut of checkpoints $S$ is Z-path-free if there is no Z-path between any two checkpoints in $S$. In particular, every checkpoint in $S$ is not in a Z-cycle.

Netzer and Xu [75] proved that a cut of checkpoints $S$ is a recovery line iff it is a Z-path-free. The following relations and operations on cuts of checkpoints are used throughout the dissertation.

Definition 6.2.5: Given two cuts of checkpoints $S_1$ and $S_2$ such that $P(S_1) \cap P(S_2) \neq \emptyset$, $S_1 \leq S_2$ if for every $p \in P(S_1) \cap P(S_2)$, $S_1|_p \xrightarrow{hb} S_2|_p$ or $S_1|_p = S_2|_p$. If $P(S_1) \cap P(S_2) \neq \emptyset$, $S_1 < S_2$ if $S_1 \leq S_2$ and $S_1 \neq S_2$ (i.e., for some $p \in P(S_1) \cap P(S_2)$, $S_1|_p \xrightarrow{hb} S_2|_p$). Also, if for every $p \in P(S_1) \cap P(S_2)$, $S_1|_p \xrightarrow{hb} S_2|_p$, then $S_1 \ll S_2$.

Definition 6.2.6: Given two checkpoints $C_{p,i}$ and $C_{p,j}$ of the same process $p$, the distance between them, denoted $dist(C_{p,i}, C_{p,j})$, is $|i - j|$. The distance between two cuts $S_1$ and $S_2$ of checkpoints such that $P(S_1) \cap P(S_2) \neq \emptyset$, denoted $dist(S_1, S_2)$, is $\max_{p \in P(S_1) \cap P(S_2)}(dist(S_1|_p, S_2|_p))$.

Definition 6.2.7: Given two cuts of checkpoints $S_1$ and $S_2$, the union of $S_1$ and $S_2$ is a cut of checkpoints $S_3$ such that for $p \in P(S_1) \cup P(S_2)$: If $p \in P(S_1) \cap P(S_2)$, then $S_3|_p$ is the latest of $S_1|_p$ and $S_2|_p$ in $p$’s local history. Otherwise, $S_3|_p = S_1|_p$ if $p \in P(S_1)$, or $S_3|_p = S_2|_p$ if $p \in P(S_2)$. Similarly, the intersection of $S_1$ and $S_2$ is a cut of checkpoints $S_3$ such that for $p \in P(S_1) \cup P(S_2)$: If $p \in P(S_1) \cap P(S_2)$, then $S_3|_p$ is the earliest of $S_1|_p$ and $S_2|_p$ in $p$’s local history. Otherwise, $S_3|_p = S_1|_p$ if $p \in P(S_1)$, or $S_3|_p = S_2|_p$ if $p \in P(S_2)$.
Definition 6.2.8: Two cycles \( \Phi_1 \) and \( \Phi_2 \) directly overlap if there is a process \( p, p \in \Phi_1 \) and \( p \in \Phi_2 \) such that \( \text{pre}_p(\Phi_2) \xrightarrow{\text{hb}} \text{post}_p(\Phi_1) \) and \( \text{pre}_p(\Phi_1) \xrightarrow{\text{hb}} \text{post}_p(\Phi_2) \). The overlapping relation is the transitive closure of direct overlap.

![Figure 6.2: Execution example with two cycle classes.](image)

For example, in Figure 6.2, the cycles \( \Phi_1 = (m_7, m_6, m_5, m_4) \) and \( \Phi_2 = (m_9, m_8) \) directly overlap because they have a common participant \( p_2 \), and \( \text{pre}_{p_2}(\Phi_2) = C_{2,2} \xrightarrow{\text{hb}} C_{2,3} = \text{post}_{p_2}(\Phi_1) \). The cycles \( \Phi_1 \) and \( (m_{10}, m_9) \) overlap since they both directly overlap with \( \Phi_2 \). In contrast, the cycle \( \Phi_3 = (m_1, m_2) \) does not overlap \( \Phi_1 \) or \( \Phi_2 \).

Overlapping partitions the cycles into equivalence cycle classes (hereafter, *cycle classes*). The cycle class of a cycle \( \Phi \) is denoted \([\Phi]\). For instance, in the execution of Figure 6.2 there are two cycle classes, \([\Phi_3]\) and \([\Phi_1]\).

Definition 6.2.9: A cycle class \( F \) is \( d \)-bounded if \( \forall \Phi \in F, \text{dist}((\text{pre}(\Phi)), \text{post}(\Phi)) \leq d \). If \( F \) is \( d \)-bounded for some \( d \), we say that \( F \) is *bounded*. If \( F \) is not bounded, then it is *unbounded*.

Given a cycle class \( F \), the set of *participants* of \( F \) is the union of participants of all the cycles in \( F \); their number is the *weight* of \( F \), denoted \( \omega(F) \). Furthermore, the *pre-border* of \( F \), denoted \( \text{pre}(F) \), is the intersection of pre-borders of all cycles in \( F \). If \( F \) is bounded, the *post-border* of \( F \), denoted \( \text{post}(F) \), is the union of post-borders of all cycles in \( F \).

It can be proved that a bounded class contains a cycle which is as “big” as the entire class. More precisely, there is a cycle \( \Phi \in F \) such that \( \text{pre}(\Phi) = \text{pre}(F) \) and \( \text{post}(\Phi) = \text{post}(F) \). For instance, in the execution of Figure 6.2 the cycle \( \Phi = (m_4, m_7, m_9, m_8, m_6, m_5) \) which is in \([\Phi_1]\) satisfies that \( \text{pre}(\Phi) = \text{pre}(\Phi) \) and \( \text{post}(\Phi) = \text{post}(\Phi) \).

Observation 1: For any cycle class \( F \), \( \text{pre}(F) \) is a Z-path-free cut. Furthermore, if \( \text{post}(F) \) exists (i.e. if \( F \) is bounded), then it is also a Z-path-free cut.

It is straightforward to verify this observation. By the definition of pre- and post-border, any Z-path that starts from a checkpoint in \( \text{pre}(F) \) (or \( \text{post}(F) \)) does not have a message with a receive event before \( \text{pre}(F) \) (or \( \text{post}(F) \)).

Claim 6.2.1 can be extended in the following way for cycle classes:
Claim 6.2.3: For every cycle class $F$ and every pair $C_{p,i}, C_{q,j}$ of checkpoints of processes $p$ and $q$ that participate in $F$, if $\text{pre}_q(F) \xrightarrow{\text{hb}} C_{q,j}$ and either $F$ is unbounded or $C_{p,i} \xrightarrow{\text{hb}} \text{post}_p(F)$, then $C_{p,i} \sim C_{q,j}$.

An execution $E$ is $d$-bounded cycles (in short, $d$-BC) if every cycle class $F \in E$ is $d$-bounded. This class is weaker than ZCF in that it allows for cycles; however, it restricts their form. In Chapter 6 we motivate this new definition.

We are now ready to explain the notion of $k$-rollback (first defined in [13]), which is used to classify executions by rollback distance and it lies at the heart of our tools for evaluating distributed checkpointing protocols. We start with some auxiliary definitions.

Definition 6.2.10: The cut of checkpoints derived from $C_{p,i}$, denoted $\text{Cut}(C_{p,i})$, is the set of the latest checkpoints from each process that occurred at or before time $t(C_{p,i})$.

Definition 6.2.11: The checkpoint interleaving level of an execution $E$, denoted $\text{IL}(E)$, is the minimal number $l$ such that for all processes $p$, for all pairs of consecutive checkpoints $(C_{p,i}, C_{p,i+1})$, no process $q \neq p$ takes more than $l$ checkpoints in the time interval $[t(C_{p,i}), t(C_{p,i+1})]$.

Notice here that since during the duration of taking a checkpoint $C_{p,i}$ by process $p$, another process may take some checkpoints. Thus, $t(C_{p,i})$ is the starting time of the checkpoint.

Definition 6.2.12: An execution $E$ is $k$-rollback for a given integer $k \geq 0$, if for every checkpoint $C_{p,i}$ there is a recovery line $R \in E$ such that $R \leq \text{Cut}(C_{p,i})$ and $\text{dist}(R, \text{Cut}(C_{p,i})) \leq k \cdot \text{IL}(E)$. If there is no $k$ such that $E$ is $k$-rollback, then $E$ is unbounded-rollback. The $k$-rollback class is the set of all $k$-rollback executions. In the following, we abuse terminology slightly by omitting the word “class”.

6.3 Classifying Executions by Rollback Distance

We introduce a classification of checkpointing patterns based on rollback distance, and discuss the degree of progress that is guaranteed by each class.

In Chapter 3 we introduce the notion of $k$-rollback. Obviously, for every $k$, $k$-rollback $\subset (k+1)$-rollback. Furthermore, executions generated by coordinated checkpointing algorithms are 1-rollback. These algorithms periodically produce a full Z-path free cut of checkpoints, $C_k = \{C_{r,k} | 1 \leq r \leq n\}$. When a failure occurs after some checkpoint $C_{p,i}$, it is always possible to roll back to $C_{i-1}$, hence the execution is 1-rollback. However, these executions are not necessarily 0-rollback. For instance, in the Chandy-Lamport protocol [34], checkpoint $C_{p,i}$ may be taken before another checkpoint $C_{q,i}$ of the same $C_i$. Similarly, not every execution produced by the sync-and-stop coordinated checkpointing algorithm [80] is 0-rollback. Since the creation of $C_i$ is not atomic, $C_{p,i}$ might be produced before $C_{q,i}$ where $p \neq q$. On the other hand, an execution of Checkpoint-After-Send [104], in which a checkpoint is taken after every send event, is 0-rollback under the assumption that sending a message and performing a corresponding checkpoint is a single atomic operation. This
is because for any checkpoint \( C_{p,i} \) and receive event that happened before \( \text{Cut}(C_{p,i}) \), the corresponding send event happened before \( \text{Cut}(C_{p,i}) \) as well. Thus, \( \text{Cut}(C_{p,i}) \) is a recovery line.

Obviously, any execution of the uncoordinated checkpointing algorithm is unbounded-rollback due to the unbounded domino effect [87]. Denote the class of executions of coordinated checkpointing algorithm as CC and the class of executions of uncoordinated checkpointing algorithm as UCC. Then we have that \( \text{CC} \subset 1\text{-rollback}, \text{CC} \not\subset 0\text{-rollback}, 0\text{-rollback} \not\subset \text{CC}, \) and UCC \( = \text{unbounded-rollback}. \)

Manivannan and Singhal showed in [73] that \( \text{SZPF} \subset \text{ZPF} \subset \text{ZCF} \). Below we relate the classification of [73] to the classification based on rollback distance.

**Theorem 6.3.1:** ZPF \( \subset 1\text{-rollback}. \)

**Proof:** Consider an execution \( E \in \text{ZPF} \) such that \( \text{IL}(E) = l \) and a checkpoint \( C_{p,i} \). We show that there is a recovery line \( R \leq \text{Cut}(C_{p,i}) \) such that \( \text{dist}(R, \text{Cut}(C_{p,i})) \leq l \). If \( \text{Cut}(C_{p,i}) \) is Z-path-free, then the theorem holds with \( R = \text{Cut}(C_{p,i}) \). Otherwise, we define a Z-path-free partial cut \( R_0 = \{ C \in \text{Cut}(C_{p,i}) \mid \exists C_1 \in \text{Cut}(C_{p,i}), C_1 \sim C \} \). \( R_0 \) is not empty, since otherwise, each checkpoint \( C \in \text{Cut}(C_{p,i}) \) has an incoming Z-path from another checkpoint in \( \text{Cut}(C_{p,i}) \), and thus, there is a Z-cycle, in contradiction to the fact that \( E \) is ZPF. Let \( P_0 = \{ p \mid p \not\in \text{Cut}(R_0) \} \) and \( T_0 = \{ q_{j} \mid q \in P_0 \) and \( R_0 \not\sim C_{q,j+1} \) and \( R_0 \cup \{ q_{j} \} \) is Z-path-free \}. The desired recovery line is \( R = R_0 \cup T_0 \).

To see why \( T_0 \) contains one checkpoint from each process in \( P_0 \), note that for each participant \( q \) of \( P_0 \), we take the latest checkpoint \( C_{q,j} \) with no Z-path from \( R_0 \). This checkpoint exists since there are no Z-paths to the initial checkpoint of \( q \). Second, if there is a Z-path from \( C_{q,j} \) to \( R_0 \), then the Z-path from \( R_0 \) to \( C_{q,j+1} \) concatenated with the Z-path from \( C_{q,j} \) to \( R_0 \) is a Z-path from \( R_0 \) to itself. This would contradict the definition of \( R_0 \), which is Z-path-free. Third, \( T_0 \) is Z-path-free, since otherwise, there are two checkpoints \( C_{q,j} \) and \( C_{s,j'} \) in \( T_0 \) such that \( C_{q,j} \sim C_{s,j'} \). Then, this Z-path appended to the Z-path from \( R_0 \) to \( C_{q,j+1} \) is a Z-path from \( R_0 \) to \( C_{s,j'} \), contradicting \( T_0 \) definition.

This proves that \( R \) is a recovery line. To complete the proof we show that \( \text{dist}(R, \text{Cut}(C_{p,i})) \leq l \). Since \( R_0 \subset R \), we only need to show that \( \text{dist}(T_0, \text{Cut}(C_{p,i})) \leq l \).

Assume, by way of contradiction, that there is a checkpoint \( C_{q,j} \in T_0 \) such that \( \text{dist}(C_{q,j}, \text{Cut}(C_{p,i})) > l \), namely, \( t(C_{q,j+1}) < t(C_{p,i}) \). Consider a checkpoint \( C_{r,m} \in R_0 \) such that \( C_{r,m} \sim C_{q,j+1} \). Since the execution is ZPF, then there is a corresponding causal path from \( C_{r,m} \) to \( C_{q,j+1} \), and thus \( t(C_{r,m}) < t(C_{q,j+1}) \). Moreover, \( C_{r,m} \) is not the last checkpoint on \( r \) because \( r \) sent a message after \( C_{r,m} \). Since \( C_{r,m} \in \text{Cut}(C_{p,i}) \), we have that \( t(C_{p,i}) < t(C_{r,m+1}) \). Therefore, we obtain that \( t(C_{r,m}) < t(C_{q,j+1}) < t(C_{q,j+1}) \leq t(C_{p,i}) < t(C_{r,m+1}) \).

Thus process \( q \) has more than \( l \) checkpoints in the interval \([t(C_{r,m}), t(C_{r,m+1})]\), contradicting the fact that \( \text{IL}(E) = l \). \( \square \)

For example, Figure 6.3 depicts a ZPF execution \( E \) where \( \text{IL}(E) = 2 \). Consider the full cut \( \text{Cut}(C_{2,3}) \in E \). As in the proof of Theorem 6.3.1, \( R_0 = \{ C_{1,3}, C_{5,2} \} \) and \( T_0 = \{ C_{2,2}, C_{3,2}, C_{4,1} \} \). Thus, the recovery line \( R \) is \( R_0 \cup T_0 = \{ C_{1,3}, C_{2,2}, C_{3,2}, C_{4,1}, C_{5,2} \} \) and \( \text{dist}(R, \text{Cut}(C_{2,3})) = 2 \).
Figure 6.3: An execution example for illustrating the proof of Theorem 6.3.1.

We showed above that $CC \not\subset 0$-rollback: Since there are no Z-paths in $CC$, $CC \subset SZPF$ and then we obtain that neither $SZPF$ nor the larger ZPF are contained in 0-rollback. Furthermore, according to Theorem 6.3.1, $ZPF \subset 1$-rollback. Therefore, $SZPF \subset ZPF \subset 1$-rollback. Figure 6.4 illustrates these relations.

Figure 6.4: Relationship between the various classes.

In the case of $d$-BC, we might need to go back from $Cut(C_{p,i})$ as far as $l \cdot (n - 1) + l \cdot (d - 1) \cdot \left(\frac{n}{2} - 1\right) + (d - 1)$ checkpoints if $n$ is an even number and $l \cdot (n - 1) + l \cdot (d - 1) \cdot \frac{n-1}{2}$ checkpoints if $n$ is an odd number to find a recovery line. This is exemplified in the execution depicted in Figure 6.5. The execution contains an even number $n$ of processes and $\frac{n}{2}$ cycles, each with two participants. The cycles form a causal sequence: a message sent just after the pre-border of every cycle in the chain arrives just before the post-border of the next cycle in the sequence. Each cycle is $d$-bounded and contains exactly $d - 1$ internal checkpoints on each participant. The interleaving of checkpoints in the execution is as shown in the figure. Checkpoint $C = C_{p,i}$ is taken so that $Cut(C_{p,i})$ is just before the most advanced checkpoint in the post-border of the first cycle in the chain.

The recovery line in the depicted scenario is a union of the pre-borders of all the cycles. The longest rollback is that of $p_n$; it is equal to the distance between $Cut(C)|_{p_n}$ and the pre-border of the cycle on $p_n$. Let us take any of the first $\frac{n}{2} - 1$ cycles, for an even $n$,
assume it is on processes $p_k$ and $p_{k+1}$. There are $l$ checkpoints on $p_n$ between each pair of consecutive cycles checkpoints of $p_k$. Furthermore, there are $l$ checkpoints on $p_n$ between the cycle pre-borders of $p_k$ and $p_{k+1}$. For the last cycle, there are $l$ checkpoints on $p_n$ between the $d$-th and $(d+1)$-th cycle checkpoints of $p_{n-1}$. In other words, we have to roll back $l$ checkpoints on $p_n$ due to each process $p_j$, $1 \leq j < n$, $(d-1) \cdot l$ checkpoints due to each cycle except the last one, and $d-1$ checkpoints due to the last cycle. Thus, the total rollback for this scenario is $l \cdot (n-1) + l \cdot (d-1) \cdot ((n/2)-1) + (d-1)$.

Below we show that the above scenario indeed incurs the longest rollback for the $d$-BC class. We start by formally proving a slightly weaker bound in Theorem 6.3.3, $l \cdot (n-1) \cdot d + (d-1)$. In the proof we partition a cut $C$ of checkpoints into disjoint subsets according to the cycle class to which each checkpoint belongs. Namely, if several checkpoints are internal to the same cycle class, they are in the same subset. On the other hand, if a checkpoint is not internal for some cycle, then it is in a subset of its own.

**Lemma 6.3.2:** If a cut $C$ of checkpoints is partitioned into subsets by cycle classes, then there is a subset with no incoming Z-paths from other subsets.

**Proof:** Partition $C$ into cycle classes, $S_1, \ldots, S_k$. Assume, by way of contradiction, that for every subset $S_i$ there is another subset $S_j$ such that $S_j \leadsto S_i$. By assumption, we can pick subsets $\{S_{i_0}, \ldots, S_{i_l}\}$ such that $S_{i_0} \leadsto S_{i_1} \leadsto \cdots \leadsto S_{i_l}$ and for every $j$, $S_{i_{j-1}} \neq S_{i_j}$, $1 \leq j \leq k$. Since there are only $k$ distinct subsets, there exist $q < l \leq k$, $l - q \geq 2$, such that $S_{i_q} = S_{i_l}$. Therefore, there is a cycle class that includes $S_{i_q}, \ldots, S_{i_l}$. This is in contradiction to the way the subsets were built.

**Theorem 6.3.3:** Consider an execution $E \in d$-BC such that $IL(E) = l$. For any checkpoint $C_{p,i}$ there is a recovery line $R \leq Cut(C_{p,i})$ such that $dist(R, Cut(C_{p,i})) \leq l \cdot (n-1) \cdot d + (d-1)$.

**Proof:** The proof is by constructing $R$. Let $\{S_1, \ldots, S_k\}$ be a partitioning of $Cut(C_{p,i})$ into subsets by cycle classes. If a subset $S_i$ contains internal checkpoints, then we replace them with the corresponding checkpoints in the pre-border of the cycle class.
Define $R_0 = \{ \cup S_i | \exists (S_j, j \neq i \land S_j \leadsto S_i) \}$. Since each subset $S_i$ is Z-path-free (by Observation 1), then $R_0$ is also Z-path-free. By Lemma 6.3.2, there is at least one subset $S_j$ that has no incoming Z-path from the others, thus $R_0 \neq \emptyset$. Also, since each cycle is $d$-bounded, the distance between checkpoints of $\text{Cut}(C_{p,i})$ and those in the pre-border of a cycle is at most $d-1$, or in other words, $\text{dist}(\text{Cut}(C_{p,i}), R_0) \leq d-1$. Let $P_0 = \{ p | p \not\in P(R_0) \}$ and $T_0 = \{ C_{q,j} | q \in P_0 \text{ and } R_0 \leadsto C_{q,j}+1 \text{ and } R_0 \cup \{ C_{q,j} \} \text{ is Z-path-free} \}$.

We iteratively extend $R_0$ to a recovery line $R$. Assume we have constructed $R_{m-1}$, $m \geq 1$. In the $m^{th}$ iteration, we add a single checkpoint to $R_{m-1}$ such that the resulting cut $R_m$ is Z-path-free. More specifically:

1. Pick some checkpoint $C_{r,j} \in T_{m-1}$ such that there are checkpoints $C \in R_{m-1}$ and $C_{r,j+x}$, $1 \leq x \leq d$, such that $C \xrightarrow{hb} C_{r,j+x}$.

2. Let $R_m = R_{m-1} \cup \{ C_{r,j} \}$.

3. Let $P_m = \{ p | p \not\in P(R_m) \}$ and $T_m = \{ C_{q,j} | q \in P_m \text{ and } R_m \leadsto C_{q,j+1} \text{ and } R_m \cup \{ C_{q,j} \} \text{ is Z-path-free} \}$.

These steps are repeated until $|R_m| = n$, that is, $R_m$ is a full cut, which is the desired recovery line $R$.

To see why $T_m$ contains one checkpoint from each process in $P_m$, note that for each participant $q$ of $P_m$, we take the latest checkpoint $C_{q,j}$ with no Z-path from $R_m$. This checkpoint exists since there are no Z-paths to the initial checkpoint of $q$. Moreover, if there is a Z-path from $C_{q,j}$ to $R_m$, then the concatenation of the Z-path from $R_m$ to $C_{q,j+1}$, with the Z-path from $C_{q,j}$ to $R_m$, is a Z-path from $R_m$ to itself. This would contradict the induction hypothesis that $R_m$ is Z-path free. Finally, $C_{q,j}$ is not in a Z-cycle: if it were, then the concatenation of the Z-path from $R_m$ to $C_{q,j+1}$ with the Z-path from $C_{q,j}$ to itself is a Z-path from $R_m$ to $C_{q,j}$.

The next lemma shows that there is a checkpoint $C_{r,j} \in T_m$ which satisfies the condition of Step 1 if $|R_m| < n$.

**Lemma 6.3.4:** For each iteration $m$, there is a checkpoint $C_{r,j} \in T_m$ such that there are checkpoints $C \in R_m$ and $C_{r,j+x}$ for some $x$, $1 \leq x \leq d$, such that $C \xrightarrow{hb} C_{r,j+x}$.

**Proof:** Consider a cut $T' = \{ C_{q,j+1} | C_{q,j} \in T_m \}$. By definition of $T_m$, $\forall C_{q,j+1} \in T'$, $R_m \leadsto C_{q,j+1}$. If there are checkpoints $C \in R_m$ and $C_{q,j+1} \in T'$ such that $C \xrightarrow{hb} C_{q,j+1}$, then the lemma trivially holds. Otherwise, partition $T'$ into cycle classes. By Lemma 6.3.2, there is some subset $S$ with no incoming Z-path from other subsets. Consider a Z-path $\xi$ from $R_m$ to some checkpoint $C_{q,j+1} \in S$. This path contains a message $msg$ from some process $s \in P(R_m)$ to some process $r \in P_m$. Note that $msg$ was sent after $R_m|_s$, otherwise the prefix of $\xi$ before $msg$ is a Z-path from $R_m$ to itself. Furthermore, it was received after $C_{r,j+1}$, otherwise $R_m|_{x \xrightarrow{hb} C_{r,j+1}}$ which contradicts the assumption above. Therefore, $msg$ was received in the checkpoint interval $I_{r,j+x}$ for some $x > 1$.  

59
Thus, \( R_m \rightharpoonup_{s} C_{r,j,x} \). To conclude the proof, it remains to be shown that \( x \leq d \). Assume by contradiction that \( x > d \). Denote \( \xi' \) the rest of \( \xi \) after \( \text{msg} \). First, note that \( r \in P(S) \), otherwise \( \xi' \) is a Z-path from \( C_{r,j,x} \) to \( C_{q,j_1} \). Next, observe that there is a Z-path from \( C_{r,j,x} \) to \( C_{r,j_1} \): if \( r = q \), then it is \( \xi' \) itself. Otherwise, both \( C_{r,j_1} \) and \( C_{q,j_1} \) are internal to the same cycle class, and there is a Z-path from \( C_{q,j_1} \) to \( C_{r,j_1} \). The concatenation of \( \xi' \) with this Z-path is a Z-path from \( C_{r,j,x} \) to \( C_{r,j_1} \). This Z-path is a cycle \( \Phi \) such that both \( C_{r,j_1} \) and \( C_{r,j,x} \) are internal to \( \Phi \). This cycle is either unbounded, or \( \text{dist}(\text{pre}_r(\Phi), \text{post}_r(\Phi)) \geq x > d \), contradicting the fact that the execution is \( d \text{-BC} \). 

To complete the proof of Theorem 6.3.3, we show the following lemma:

**Lemma 6.3.5:** For every \( m \geq 0 \) and every non-virtual checkpoint \( C \in R_m \),
\[
\text{dist}(\text{Cut}(C), \text{Cut}(C_{p,i})) \leq |R_m| \cdot l \cdot d.
\]

**Proof:** For \( m = 0 \), we pointed out that \( \text{dist}(R_0, \text{Cut}(C_{p,i})) \leq d - 1 \). Fix an arbitrary non-virtual checkpoint \( C_{q,j} \in R_0 \), and consider the checkpoint \( C_{q,h} = \text{Cut}(C_{p,i})_q \). \( C_{q,h} \) cannot be virtual because in this case \( C_{q,h} = C_{q,j} \) according to the construction of \( R_0 \). Since \( C_{q,j} \) belongs to \( R_0 \), \( \text{dist}(C_{q,j}, C_{q,h}) \leq d - 1 \). Since \( IL(E) = l \), \( \forall j', j \leq j' \leq h \), \( \text{dist}(\text{Cut}(C_{q,j}), \text{Cut}(C_{q,j'})) \leq l \).

Thus, we have that \( \text{dist}(\text{Cut}(C_{q,j}), \text{Cut}(C_{q,h+1})) \leq d \cdot l \). Since \( C_{q,h} \in \text{Cut}(C_{p,i}) \), then \( \text{Cut}(C_{p,i}) < \text{Cut}(C_{q,h+1}) \). Therefore,
\[
\text{dist}(\text{Cut}(C_{q,j}), \text{Cut}(C_{p,i})) \leq \text{dist}(\text{Cut}(C_{q,j}), \text{Cut}(C_{q,h+1})) \leq d \cdot l
\]

This proves the induction base. Assume now that the inequality holds for iteration \( m \geq 0 \), we prove that it holds for iteration \( m + 1 \), in which we extend \( R_m \) with checkpoint \( C_{q,j} \) such that there is a checkpoint \( C \in R_m \) and \( C \rightharpoonup_{b} C_{q,j+x} \) for some \( x \), \( 1 \leq x \leq d \). Therefore, \( \text{Cut}(C) < \text{Cut}(C_{q,j,x}) \) Similarly to the induction base, we have that
\[
\text{dist}(\text{Cut}(C_{q,j}), \text{Cut}(C_{q,j,x})) \leq l \cdot l \leq d \cdot l
\]

Otherwise, \( \text{dist}(\text{Cut}(C_{q,j,x}), \text{Cut}(C_{p,i})) \leq \text{dist}(\text{Cut}(C), \text{Cut}(C_{p,i})) \leq |R_m| \cdot l \cdot d \), and
\[
\text{dist}(\text{Cut}(C_{q,j}), \text{Cut}(C_{p,i})) \leq d \cdot l + |R_m| \cdot l \cdot d = (|R_m| + 1) \cdot l \cdot d = |R_{m+1}| \cdot l \cdot d
\]

To conclude the theorem, we prove that for every checkpoint \( C_{q,j} \in R \), \( \text{dist}(C_{q,j}, \text{Cut}(C_{p,i})_q) \leq l \cdot (n - 1) \cdot d + (d - 1) \). If \( C_{q,j} \) was added in first iteration, then \( C_{q,j} \in R_0 \) and \( \text{dist}(C_{q,j}, \text{Cut}(C_{p,i})_q) \leq \text{dist}(R_0, \text{Cut}(C_{p,i})) \leq d - 1 \leq l \cdot (n - 1) \cdot d + (d - 1) \).

Otherwise, \( C_{q,j} \) was added at some iteration \( m \), \( 1 \leq m \leq n - 1 \). Note that \( |R_0| \geq 1 \) and since we add a checkpoint in each iteration, we have at most \( n - 1 \) iterations. By Lemma 6.3.4, there is a checkpoint \( C \in R_m \) such that \( C \rightharpoonup_{b} C_{q,j+x} \) for some \( x \), \( 1 \leq x \leq d \). Since \( C \) cannot be virtual, \( \text{dist}(\text{Cut}(C), \text{Cut}(C_{p,i})) \leq |R_m| \cdot d \cdot l \) by Lemma 6.3.5. If \( C_{q,j} \rightharpoonup_{b} C_{q,j+x} \), then
\[
\text{dist}(C_{q,j}, \text{Cut}(C_{p,i})_q) \leq \text{dist}(C_{q,j}, C_{q,j+x}) = x \leq d \leq (d - 1) + (n - 1) \cdot d \cdot l
\]
We can strengthen this result by optimizing the way the recovery line is constructed. In the proof, we add a single checkpoint to the pre-border to need to rollback due to at most condition of Step 1 holds since we add a single checkpoint to the pre-border to need to rollback due to at most condition of Step 1 holds since

\[
\mathrm{dist}(C_{q,i}, \text{Cut}(C_{p,i})|_q) = \mathrm{dist}(C_{q,j}, \text{Cut}(C_{p,i})|_q) + \mathrm{dist}(C_{q,j+1}, \text{Cut}(C_{p,i})|_q) \leq d + \mathrm{dist}(C_{q,j+d}, \text{Cut}(C_{p,i})|_q) \leq d + (\mathrm{dist}(\text{Cut}(C)|_q, \text{Cut}(C_{p,i})|_q) - 1) \leq (d - 1) + |R_m| \cdot d \cdot l \leq (d - 1) + (n - 1) \cdot d \cdot l
\]

Figure 6.6 depicts a 2-BC execution \(E\) with \(\text{IL}(E) = 2\). Consider the full cut \(\text{Cut}(C_{2,3})\). According to the proof of Theorem 6.3.3, the partitioning of \(\text{Cut}(C_{2,3})\) into cycle classes includes three sets of checkpoints \(\{S_1, S_2, S_3\}\), where \(S_1 = \{C_{1,3}, C_{2,2}\}\), \(S_2 = \{C_{3,3}, C_{4,2}\}\), and \(S_3 = \{C_{5,3}\}\). Since \(S_3 \sim S_2\), then in the first iteration we obtain that \(R_0 = S_1 \cup S_2 = \{C_{1,3}, C_{2,3}, C_{5,3}\}\) and \(T_0 = \{C_{3,1}, C_{4,2}\}\). According to the condition of Step 1, since \(C_{5,3} \in R_0\) and \(C_{5,3} \cong_{\text{hb}} C_{4,3}\), then we obtain in the second iteration that \(R_1 = R_0 \cup \{C_{4,2}\}\) and \(T_1 = \{C_{3,1}\}\). Lastly, in the third iteration we obtain that \(R_2 = R_1 \cup \{C_{3,1}\}\). Notice that the condition of Step 1 holds since \(C_{4,2} \cong_{\text{hb}} C_{3,2}\).

![Figure 6.6: An execution example for illustrating the proof of Theorem 6.3.3.](image)

Since \(l \cdot (n - 1) \cdot d + (d - 1) < l \cdot n \cdot d\), Theorem 6.3.3 implies that \(d\)-BC \(\subset (n \cdot d)\)-rollback. We can strengthen this result by optimizing the way the recovery line is constructed. In the proof, we add a single checkpoint to \(R_m\) in each iteration \(m \geq 1\). Instead, if this checkpoint were in a pre-border of a cycle class, we could add all the checkpoints in the pre-border to \(R_m\). Note that a cycle class contains at least two participants, so we would need to rollback due to at most \(\left\lceil \frac{n}{2} \right\rceil\) cycles while constructing the recovery line. If the number of processes \(n\) is odd, we would need to rollback at most \(l \cdot (d - 1)\) checkpoints due to every cycle, and not in every iteration as in the proof above. If \(n\) is even, the longest rollback will be \(l \cdot (d - 1)\) due to every cycle except the last one, and \(d - 1\) due to the last cycle. Taking into consideration that we might also need to roll back \(l\) checkpoints \(n - 1\) times due to interleaving of checkpoints in the execution as in the above proof, we obtain the following result:

**Claim 6.3.6:** Consider an execution \(E \in d\)-BC such that \(\text{IL}(E) = l\). For any checkpoint \(C_{p,i}\) there is a recovery line \(R \leq \text{Cut}(C_{p,i})\) such that

61
• for even \( n \), \( \text{dist}(R, \text{Cut}(C_{p,i})) \leq l \cdot (n - 1) + (d - 1) \cdot (l \cdot \left(\frac{n}{2} - 1\right) + 1) \);

• for odd \( n \), \( \text{dist}(R, \text{Cut}(C_{p,i})) \leq l \cdot (n - 1) + (d - 1) \cdot l \cdot \frac{n-1}{2} \).

For even \( n \geq 2 \), \( l \cdot (\frac{n}{2} - 1 + 1) \leq l \cdot (n - 1) \). For odd \( n \geq 1 \), \( \frac{n-1}{2} \leq n - 1 \). In both cases, \( \text{dist}(R, \text{Cut}(C_{p,i})) \leq l \cdot (n - 1) + (d - 1) \cdot l \cdot (n - 1) = l \cdot (n - 1) \cdot d \). Which implies that:

**Corollary 6.3.7:** For any \( d > 0 \), \( d\text{-BC} \subset ((n - 1) \cdot d)\text{-rollback} \).

Note that ZCF is equivalent to 1-BC. This is because an execution \( E \in 1\text{-BC} \) has no cycles with internal checkpoints. By Claim 6.2.1, every internal checkpoint is in a Z-cycle, and thus \( E \) has no Z-cycles. Therefore, \( E \in \text{ZCF class} \). Claim 6.3.6 implies that:

**Corollary 6.3.8:** \( \text{ZCF} \subset (n - 1)\text{-rollback} \).

### 6.4 A \( d\text{-BC} \) Checkpointing Protocol

In this section, we present a checkpointing protocol that generates only \( d\text{-BC} \) executions, for a given parameter \( d \). The protocol is optimistic: processes take checkpoints independently allowing cycles to be formed, as long as no cycle has more than \( d \) checkpoints. When a cycle class \( F \) of size \( d \) is generated, the protocol performs the Chandy-Lamport (C-L) coordinated checkpointing algorithm among the participants of \( F \) in order to guarantee \( d\)-boundness. The delivery of application messages from processes not in \( F \) is delayed during the C-L algorithm. It should be emphasized that the C-L algorithm is performed only among participants of \( F \) and not all the processes. If more participants are added to \( F \), then the C-L algorithm is restarted among the new set of participants; however, this may happen at most \( n - 1 \) times.

A distinguishing feature of this protocol is that it does not use message piggybacking. Therefore, it does not have the drawback of paying the overhead of piggybacking information on top of application messages (see [18] for a discussion of this problem).

Each process \( p \) maintains an up-to-date recovery line \( R_p \). It also maintains a dependency graph whose nodes represent checkpoint intervals and whose edges specify existing dependencies: When a message is sent during \( I_{p,i} \) and received in \( I_{q,j} \), a directed edge is drawn from \( I_{p,i} \) to \( I_{q,j} \). This dependency graph is similar to the rollback-dependency graph [25] and the checkpoint graph [103] except that nodes in our dependency graph represent checkpoint intervals rather than checkpoints. We present more implementation details of the \( d\text{-BC} \) protocol in Starfish in Appendix B.

#### 6.4.1 Data Structures

Each process maintains the following data structures:

**\( R_p \)** - An \( n \)-vector of integers, the recovery line maintained by process \( p \). \( R_p[q] = k \) indicates that \( C_{q,k} \in R_p \). Initially, \( R_p[q] = 0 \), for every \( q \).
$L_p$ - An $n$-vector of integers. $L_p[q], 1 \leq q \leq n$, indicates the latest checkpoint of process $q$ known to process $p$. $L_p$ is a full cut of checkpoints, and $L_p[q] \geq R_p[q]$ for every process $q$. Initially, $L_p[q] = 0$, for every $q$.

$M_p$ - A matrix of size $n \times n$, which contains the dependency graph. Each entry in the matrix is a set of integer pairs; $(i, j) \in M_p[p][q]$ indicates that there is a message sent in checkpoint interval $I_{q,j+1}$ and received in checkpoint interval $I_{p,i}$. We only maintain dependencies between checkpoint intervals after $R_p$. If $x = (i, j) \in M_p[p][q]$, then $x$.from refers to $j$, and $x$.to refers to $i$. Initially, all entries of $M_p$ are empty.

$S_p$ - An $n$-vector of sets of process id’s. A non-empty entry $S_p[q]$ indicates that $q$ is in a tail of a cycle class $F$, and $q$ has taken at least $d$ checkpoints since receiving a message from a participant of $F$. In this case, $S_p[q]$ is the union of participants of all such cycle classes. Note that receiving a message from $q$ may create a cycle which is not $d$-bounded.

The protocol defines a control message for information exchange among the processes upon taking a new local checkpoint.

6.4.2 The Protocol

Three types of events may occur at process $p$: receiving an application message from another process, taking a local independent checkpoint, and receiving a control message. Below we present the protocol by describing the actions it takes in response to each of these events.

**Receiving an Application Message**

Send/receive events determine dependencies among checkpoint intervals of different processes. A message sent in interval $I_{q,j+1}$ and received in interval $I_{p,i}$ creates a dependency between these intervals. $M_p$ is updated in the following way: If $M_p[p][q]$ already contains a pair $x$ with $x$.to = $i$, then $x$.from is updated to $j$. Otherwise, the pair $(i, j)$ is added to $M_p[p][q]$.

If $p$ receives a message from $q$ and $S_p[q] \neq \emptyset$, then delivering this message may create a cycle which is not $d$-bounded. Thus, process $p$ invokes the C-L algorithm among the set of processes in $S_p[q]$ before delivering the message.

**Taking a Local Checkpoint**

When process $p$ takes a checkpoint $C_{p,i}$, it finds all the checkpoints that $C_{p,i}$ directly depends on. Then, $p$ broadcasts these checkpoints to all processes (including itself). More precisely, process $p$ performs broadcast($Cut$), where $Cut = \{C_{q,j} | (i, j) \in M_p[p][q], 1 \leq q \leq n\}$.

**Receiving a Control Message**

Message $\langle Cut \rangle$ from $q$ after taking $C_{q,j}$, it performs the following steps:
1. \( L_p[q] = L_p[q] + 1. \) \{By the FIFO assumption, \( j = L_p[q] + 1. \}\)

2. If \( p \neq q \), \( p \) updates its dependency graph \( M_p \) as explained above (upon receiving an application message).

3. \( p \) tries to advance \( R_p \) by calling the function \textbf{tryToAdvance} (Figure 6.7).

4. Compute \( S_p \).

\begin{verbatim}
recursionDepth = 0 \{Initialization\}

\textbf{tryToAdvance}(C_{q,j})
1: \quad \text{recursionDepth} = \text{recursionDepth} + 1

2: \quad \{ \text{Check if } q \text{ is a participant in a cycle } \Phi: \} \quad (preF, postF) = \text{findClassBorders}(C_{q,j}).

3: \quad \text{If } (\text{recursionDepth} = 1) \text{ and } (d(C_{q,j}, preF[q]) = d), \text{ then:}
\quad \quad \text{performs C-L coordinated checkpointing}
\quad \quad \text{protocol on } P(preF).
\quad \quad \{\text{This operation produces the post-border of the cycle and updates } postF.\}

4: \quad \text{If } (postF = \emptyset) \text{ or } (\exists C_{w,k} \in R_p \text{ such that } w \notin P(postF) \text{ and } C_{w,k} \sim postF),
\quad \quad \text{then return FAILURE.}

5: \quad \text{P} = \emptyset
\quad \{\text{Advance } R_p \text{ to include } postF:}\}
\quad \text{For each checkpoint } C \in postF,
\quad \quad \text{let } \text{P} = \text{P} \cup \text{advancement}(C).
\quad \text{For each process } v \in \text{P}
\quad \quad \text{For } k = L_p[v] \text{ downto } (R_p[v] + 1)
\quad \quad \text{If } (\text{tryToAdvance}(C_{v,k}) = \text{SUCCESS}) \{\text{recursive call}\}
\quad \quad \quad \text{break out of the inner loop.}
\quad \text{return SUCCESS.}
\end{verbatim}

Figure 6.7: Process \( p \) tries to advance \( R_p \) to include checkpoint \( C_{q,j} \).

The recursive function \textbf{tryToAdvance} (Step 2 in Figure 6.7) starts by invoking the function \textbf{findClassBorders} which finds the pre- and post-borders of the candidate checkpoint. This function takes a single checkpoint as an argument. If the given checkpoint is internal to a cycle or it belongs to the post-border of a cycle class, then \textbf{findClassBorders} returns the pre- and post-borders of the cycle class to which the checkpoint belongs. Otherwise, the function returns a singleton set consisting of the checkpoint itself. In Section 6.4.3, we elaborate on the implementation of \textbf{findClassBorders}.
In Step 3 of Figure 6.7, if this is the first level of the recursion, and the candidate checkpoint is internal to a cycle class $F$, and $\text{dist}(C_q,j,\text{pre}(F)|q) = d$, then $\text{tryToAdvance}$ performs the C-L coordinated checkpointing protocol on $P(F)$ to ensure that $F$ is $d$-bounded. By the C-L protocol [34], after a process $p \in P(F)$ takes a checkpoint, it sends a marker to all the other in $P(F)$. We modify the C-L protocol such that process $p$ sends with the marker a cut that contains all the checkpoints that the new forced checkpoint directly depends on. On the other hand, whenever a process $q \in P(F)$ receives the marker, it updates its dependence graph $M_q$ accordingly.

Thus, when $\text{tryToAdvance}$ reaches Step 4, three cases are possible:

1. The candidate checkpoint was not in a cycle. In this case, the variable $\text{post}F$ contains only the checkpoint itself.

2. The candidate checkpoint is in a cycle whose post-border is formed. In this case, the variable $\text{post}F$ contains the post-border of the cycle class.

3. The post-border of the cycle class to which the candidate checkpoint is internal is not known yet. In this case, the variable $\text{post}F$ contains an empty set.

In all cases, $\text{tryToAdvance}$ checks whether $R_p$ can be advanced to $\text{post}F$. It cannot be advanced if $\text{post}F$ depends on another cycle class in which case the function returns. Otherwise, $\text{tryToAdvance}$ advances $R_p$ to $\text{post}F$ by applying the function $\text{advancement}$ presented in Figure 6.8.

First, $\text{advancement}$ updates $R_p[q]$ to include $\text{post}F$. Then, it removes all the dependencies in $M_p$ that depend on intervals $I_{q,k}$, $k \leq j$, and remembers the processes whose checkpoints depend on $I_{q,k}$, $k \leq j$. Finally, it returns the set of processes that are now candidates for advancement. When $\text{advancement}$ returns, the protocol tries to advance the recovery line on all the returned candidate processes by recursively calling $\text{tryToAdvance}$.

**Figure 6.8:** Process $p$ advances $R_p$ to include checkpoint $C_{q,j}$.

If $R_p$ was advanced after the last call to $\text{tryToAdvance}$, then the protocol modifies $S_p$ to represent the updated knowledge about the tails.
6.4.3 Computing an Overlapping Class

As explained in the previous section, each process maintains a dependency graph associated with the execution. In this graph nodes represent checkpoint intervals and edges correspond to the messages sent between those intervals. We now define a slightly different cycle graph, which is obtained from dependency graph by adding edges between the nodes corresponding to adjacent checkpoint intervals on the same process (called “interval” edges to distinguish them from “message” edges). For example, Figure 6.9 depicts the cycle graph for the execution of Figure 6.2.

Obviously, an execution contains a Z-path from $C_{p,i}$ to $C_{q,j}$ if and only if its cycle graph contains a path from $I_{p,i+1}$ to $I_{q,j}$ with at least one message edge. Consequently, cycles in the execution correspond to cycles in the cycle graph. Furthermore, the set of cycle classes in the former is isomorphic to the set of non-elementary strongly connected components of the latter. For example, we showed in Section 3.1 that there are only two cycle classes in the execution of Figure 6.2, and indeed, the cycle graph of Figure 6.9 contains two strongly connected components.

This observation is used by the mechanism of finding an overlapping class with its borders. Specifically, the function `findClassBorders` finds a strongly connected component to which the checkpoint interval $I_{p,i}$ belongs. Finding strongly connected components of a directed graph is a well studied problem [38], and therefore we do not present its detailed implementation. It should be emphasized that processes do not need to maintain the cycle graph explicitly because it can be easily computed out of the dependency graph.

Note that tails correspond to the outgoing paths from strongly connected components. This fact is used to update $S_p$ when a control message is received.

6.4.4 Protocol Properties

In this section, we prove the safety and progress of the $d$-BC protocol. It can be shown that the union of $R_p$’s of all the processes is a recovery line. This recovery line $R$ can be computed during recovery: $\forall i, 1 \leq i \leq n, R[i] = \max_{1 \leq p \leq n} (R_p[i])$. 

Figure 6.9: The cycle graph for the execution of Figure 6.2. This graph contains two strongly connected components.
Lemma 6.4.1: For every cycle $\Phi$ and recovery line $R$, either $R \leq \text{pre}(\Phi)$ or $\text{post}(\Phi) \leq R$.

Proof: Assume, by way of contradiction, that there is recovery line $R$, and two participants $p$ and $q$ of a cycle $\Phi$ such that $\text{pre}_p(\Phi) \not\leq R|_p$ and $R|_q \not\leq \text{post}_q(\Phi)$. By Claim 6.2.1, there is a Z-path from $R|_q$ to $R|_p$, implying that $R$ is not a recovery line.

The following lemma justifies the function $\text{tryToAdvance}$.

Lemma 6.4.2: Given recovery line $R$ and bounded cycle class $F$ such that $\text{pre}(F) \subseteq R$, either $(R \setminus \text{pre}(F)) \cup \text{post}(F)$ is a recovery line, or there is a Z-path from $R \setminus \text{pre}(F)$ to $\text{post}(F)$.

Proof: Let $F$ be a cycle class such that $\text{post}(F)$ exists and $\text{pre}(F) \subseteq R$, where $R$ is a recovery line. According to Claim 1 $\text{pre}(F)$ and $\text{post}(F)$ are Z-path-free cuts. Furthermore, if there is no Z-path from $R \setminus \text{pre}(F)$ to $\text{post}(F)$, then $R' = R \setminus \text{pre}(F) \cup \text{post}(F)$ is a recovery line such that $R < R'$.

The following claim expresses the safety property of the protocol:

Lemma 6.4.3: For every process $p$, $R_p$ is always recovery line.

Proof: $R_p$ is set to the initial recovery line at the beginning of the protocol. The only place where $R_p$ is modified is in $\text{tryToAdvance}$ in Step 5 of Figure 6.7. This only occurs if $\text{post}(F)$ exists, for a cycle class $F$, and there is no Z-path from $R_p \setminus \text{pre}(F)$ to $\text{post}(F)$. By Lemma 6.4.2, $R_p$ remains a recovery line after this modification.

The progress property of the protocol stems from the results of Section 6.3, since Step 3 of $\text{tryToAdvance}$ ensures that every cycle class is $d$-bounded.

Theorem 6.4.4: An execution $E$ generated by the protocol is $d$-BC.

Proof: We show that for every cycle class $F$, $\text{dist}(\text{pre}(F), \text{post}(F)) \leq d$. We examine the cases where the distance between $\text{pre}(F)$ and $\text{post}(F)$ can exceed $d$. Since cycles are generated by application messages and the distance between pre- and post-borders is measured in the number of local checkpoints, then $F$ may exceed the bound $d$ only due to receiving an application message or taking a local checkpoint.

Upon taking a new checkpoint $C_{q,j}$, $q$ broadcasts a control message as described in Section 6.4.2. Then, upon receiving the control message each process applies $\text{tryToAdvance}$ on $C_{q,j}$. In Step 2 of Figure 6.7, $\text{tryToAdvance}$ finds the cycle class $F$ of $C_{q,j}$ and its pre-border. Then, it invokes the C-L snapshot protocol to produce $\text{post}(F)$ if the distance between $\text{pre}(F)$ and $C_{q,j}$ is $d$ (Step 3 of Figure 6.7). Thus, in this case the $d$-BC protocol ensures that upon taking a new checkpoint all the cycle classes are still $d$-bounded.

On the other hand, upon receiving an application message a cycle class $F$ can be generated such that $\text{dist}(\text{pre}(F), \text{post}(F)) > d$. This can happen only if a process $p$ receives an application message from a process $q \in \text{tail}(F)$ and $q$ has taken more than $d$ checkpoints since receiving a message from a participant of $F$. Consequently, if $p$ sends a message to a participant of $F$, then $q$ becomes a participant of $F$ where the distance between $\text{pre}(F)|_q$...
and the latest checkpoint of $q$ is more than $d$ implying that $dist(pre(F), post(F)) > d$. The $d$-BC protocol deals with this problem by having each process $p$ maintain a data structure $S_p$ of all such processes as described in Section 6.4.1. Hence, if $p$ receives an application message from such a process $q$ (indicated as $S_p[q] \neq \emptyset$), $p$ invokes the C-L algorithm among $F$’s participants before delivering the message to ensure that $F$ is $d$-bounded. □

By Corollary 6.3.7, $d$-BC $\subset ((n - 1) \cdot d)$-rollback. In other words, during a recovery, there is no need to roll back too far to find a recovery line. Furthermore, we claim that this recovery line can be efficiently calculated based on the protocol knowledge. Specifically, if there is no need to roll back too far to find a recovery line. Furthermore, we claim that this recovery line is taken. By the

**Lemma 6.4.5:** Given an execution $E$, for any process $p$, if at some point in $E$ there is a recovery line $R'_p$ such that $R_p < R'_p \leq L_p$, then $R_p$ is advanced to include $R'_p$.

**Proof:** Assume, by way of contradiction, that there is a recovery line $R'_p$ such that $R_p < R'_p \leq L_p$, yet $R_p$ is not advanced to include $R'_p$, and consider the first time this happens.

By the code of the $d$-BC protocol, $p$ invokes tryToAdvance (of Figure 6.7) for each control message it receives with the corresponding checkpoint $C_{o,j}$ as a parameter. Also, by the code, an invocation of tryToAdvance may fail to advance $R_p$ to include the checkpoints of $post(F)$, for some cycle class $F$ such that $R_p \leq pre(F)$ and $post(F) \leq R'_p$, if one of the following two conditions hold (these conditions are checked in Step 4 of Figure 6.7):

a. $p$ did not receive the control message corresponding to some checkpoint in $post(F)$, or

b. there is a checkpoint $C_{w,k} \in R_p$ such that $w \not\in P(post(F))$ and $C_{w,k} \rightarrow post(F)$.

Clearly, since $R'_p$ is a recovery line, the post-border of each such cycle class exists, tryToAdvance cannot fail due to Condition (a). We show below that Condition (b) does not hold for some cycle class $F$ such that $post(F) \subseteq R'_p$, which leads to a contradiction.

By the minimality of $R'_p$, Condition (b) holds for every cycle class between $R_p$ and $R'_p$. Consider a cut of checkpoints $Q_p = \{C_{q,j} \mid C_{q,j-1} \in R_p \land (C_{q,j} \xrightarrow{hb} R'_p[q] \lor C_{q,j} = R'_p[q])\}$. Intuitively, $Q_p$ is the set of first checkpoints after $R_p$ that are not later than $R'_p$. Since $R_p < R'_p$, $Q_p$ is not empty. We partition the checkpoint intervals between $R_p$ and $Q_p$ into disjoint subsets $I = \{I_1, \cdots, I_m\}$ according to the cycle classes to which the checkpoints of
that all intervals between $R_p$ and $Q_p$ whose corresponding checkpoints in $Q_p$ belong. That is, all intervals between $R_p$ and $Q_p$ whose corresponding checkpoints in $Q_p$ are in the same cycle class belong to the same subset $I_i$. On the other hand, if a checkpoint is not internal to any cycle class, then the corresponding checkpoint interval is a singleton subset. Figure 6.10 depicts an execution example where $\mathcal{I} = \{I_1, I_2, I_3, I_4\}$.

By Condition (b), for every $I_i \in \mathcal{I}$, there is a $Z$-path from another subset $I_j \in \mathcal{I}$. Thus, there are at least two subsets of $\mathcal{I}$ that belong to the same cycle class, which contradicts the above partition. This implies that Condition (b) does not hold for some cycle class $F$ between $R_p$ and $R'_p$. By the minimality of $R'_p$, $\text{post}(F) \subseteq R'_p$ and the checkpoint $C_{o,j} \in \text{post}(F)$.

So, \textbf{tryToAdvance} succeeds in advancing $R_p$ to include $\text{post}(F)$. Moreover, after $R_p$ is advanced, the function \textbf{advancement} returns all processes whose checkpoints belong to some $F'$ such that $F \sim F'$. Following this, \textbf{tryToAdvance} is called recursively for each of them. However, now neither Condition (a) nor (b) holds for each of these recursive invocations of \textbf{tryToAdvance}, and by recursion, eventually $R_p$ is advanced to include all checkpoints that appear in any of the cycle classes $F$ such that $\text{post}(F) \subseteq R'_p$. Moreover, by the minimality of $R'_p$, there are now no cycle classes between $R_p$ and $L_p$. A contradiction.

Figure 6.10: An execution example with a partition $\mathcal{I} = \{I_1, I_2, I_3, I_4\}$ of the checkpoint intervals between $R_p$ and $Q_p$.

Theorem 6.4.6: Given an execution $E$ and a recovery line $K \in E$, denote $t_K = \max_{1 \leq p \leq n}(t(K|_p))$. If the message delivery delay of the system is bounded by $\delta$, then for every process $p$, $K \leq R_p$ at time $t_K + \delta$.

Proof: Let $p$ be a process, and let $t'$ be the time in which $p$ receives the last control message corresponding to any of the checkpoints in $K$. Clearly, $t' \leq t_K + \delta$, and by time $t'$, $p$ received the control messages of all checkpoints belonging to $K$. Moreover, by Observation 2, we have that $K \leq L_p$ at time $t'$. Thus, by Lemma 6.4.5, $K \leq R_p$ at time $t'$.

69
Chapter 7

Evaluating Distributed Checkpointing Protocols

7.1 Overview

There are several schemes proposed in the literature for evaluating distributed checkpointing protocols. However, none of them address the rollback propagation that dramatically affects the protocols’ performance. In this section, we present a technique called overhead ratio for evaluating distributed checkpointing protocols. This technique gives a quantitative evaluation for each checkpointing protocol that allows for easy comparison of checkpointing protocols. At the end of this chapter, we present several checkpointing protocols and compare them using the overhead ratio technique.

7.2 Definitions and Notations

Definition 7.2.1: The checkpoint overhead, denoted o, is the increase in the execution time of a process p because a single checkpoint.

Definition 7.2.2: The checkpoint latency, denoted L, is the duration of time required to take a single checkpoint.

Clearly, o \leq L in every checkpointing mechanism [98]. For example, in sequential checkpointing [82], a process suspends normal execution in order to save its state on stable storage, and thus o = L. In contrast, in forked checkpointing [17, 81], the process forks a child process to save its state and continues normal execution concurrently, and thus o < L. This situation is illustrated in Figure 7.1(b).

Definition 7.2.3: The recovery overhead, denoted r, is the duration of time required to restart a process from a given local checkpoint. It depends on the restart mechanism but not on the checkpointing mechanism. We stress that r does not contain the time during which the failed process was down.
For the purpose of evaluation, we further assume that every process takes an independent checkpoint every $T$ units of time. However, the time between any two consecutive checkpoints might be shorter because forced checkpoints can occur at arbitrary times. Figure 7.1(a) shows an example where there is one forced checkpoint ($C_{p,i-1}$) between two consecutive independent checkpoints ($C_{p,i-2}$ and $C_{p,i}$).

![Figure 7.1](image.png)

Figure 7.1: Specifying the parameters for the events in process $p$.

Many distributed checkpointing protocols produce control overhead [44]. Control overhead is the overhead due to control information. We denote by $\text{expctMsgsNum}(P)$ the expected number of control messages in a checkpoint interval, and by $\text{expctMsgsSize}(P)$ the expected total size of the control information for a checkpoint interval, both with respect to a given checkpointing protocol $P$. The expected control overhead of protocol $P$ is therefore $M(P) = \text{expctMsgsNum}(P) \cdot w_m + \text{expctMsgsSize}(P) \cdot w_b$, where $w_m$ is the “setup” time for sending a message, and $w_b$ is the additional per-bit delay associated with sending a message.

The total checkpoint overhead, denoted $O$, is the increase in the execution time of a process $p$ because of a checkpoint $C_{p,i}$ and the control overhead corresponding to $C_{p,i}$. Namely, $O = o + M$.

If the control information is piggybacked on application messages, then the communication-pattern of the execution determines the control overhead. Therefore, for some executions we need to determine the message rate, as defined below, to compute the control overhead.

**Definition 7.2.4:** Given an execution $E$, the message rate of $E$, denoted $MR(E)$, is the expected number of data messages sent in a checkpoint interval $I_{p,i} \in E$.

Netzer and Xu [75] claimed that if a checkpoint is in a Z-cycle, then it is not part of a recovery line, namely, it is useless. However, not every checkpoint, which is not in a Z-cycle, is part of a recovery line.

**Definition 7.2.5:** Given an execution $E$ and a checkpoint $C \in E$. $C$ is called exploited checkpoint if there is a recovery line $R \in E$ such that $C \in R$. 

71
Given an execution $E$, we assume that there is a recovery mechanism that finds the most advanced recovery line in $E$. A recovery line $R$ is said to be the most advanced recovery line if there is no other recovery line $R'$ such that $R \leq R'$. Intuitively, if a process $p$ has recovered from a checkpoint $C_{p,i}$, then upon further failure $p$ can only recover from a checkpoint $C_{p,j}$ for $j \geq i$. In this paper we only consider executions with recovery mechanisms that find the most advanced recovery line.

### 7.3 The Overhead Ratio

Consider an execution $E \in k$-rollback, and a process $p \in E$. Suppose that $p$ is running in the checkpoint interval $I_{p,i+1}$, namely, it has taken the checkpoint $C_{p,i}$ but not yet $C_{p,i+1}$. If a failure occurs during $I_{p,i+1}$, then $p$ needs to recover from the newest exploited checkpoint. By Chapter 6, since $E \in k$-rollback, the newest exploited checkpoint is no more than $k \cdot IL(E)$ checkpoints backwards. Notice here that since we only consider recovery mechanisms that always find the most recent recovery line, if $p$ has recovered from $C_{p,l}$ for $i - k \leq l \leq i$, then $p$ will not recover from $C_{p,l-j}$, for any $j > 0$, if it encounters additional failures.

Recall that $T$ is the duration between the establishment of two consecutive independent checkpoints as depicted in Figure 7.1. Consider an execution $E$ with a checkpointing protocol $\mathcal{P}$. Denote by $F(\mathcal{P})$ the expected number of forced checkpoints that occur during $T$ when there are no failures, then the expected time of a checkpoint interval $I_{p,i} \in E$ is $\tau(\mathcal{P}) = \frac{T}{F(\mathcal{P})+1}$. Notice here that $\tau$ includes the local checkpoint overhead and the control overhead. Let $\tau'$ be the expected time of a checkpoint interval without the checkpoint and control overheads, namely, $\tau' = \tau - (o + M) = \tau - O$. In other words, $\tau'$ is the amount of useful computation events performed in a checkpoint interval. Figure 7.1(a) depicts an example where the expected number of forced checkpoints is 1 ($F = 1$) and thus the expected duration of a checkpoint interval $I_{p,i}$ is $\tau = \frac{T}{2}$.

On the other hand, if one or more failures occur during $I_{p,i}$, then the expected time of $I_{p,i}$ is more than $\tau$. For instance, in Figure 7.2 there is a failure during $I_{p,i+1}$. After a failure, process $p$ must rollback to the latest exploited checkpoint, incurring $r$ units of time due to recovery. Moreover, after the rollback, $L - o$ units of computation that were performed during the checkpoint latency should be performed again. This is necessary because the computation that happened concurrently with the checkpoint is not part of the saved data [84, 98]. Therefore, in the presence of one failure, $\tau + r + (L - o)$ units of time are required to complete the checkpoint interval.

The overhead ratio was first defined by Ziv and Bruck [110] and used by Vaidya [98] for performance analysis. We augment their definition to our generalized model as follows (see also [14]).

**Definition 7.3.1:** Consider an execution $E \in k$-rollback with a checkpointing protocol $\mathcal{P}$. Let $\Gamma_k(\mathcal{P})$ be the expected execution time of a checkpoint interval $I_{p,i} \in E$ and let $\tau(\mathcal{P}) = \frac{T}{F(\mathcal{P})+1}$ and $\tau'(\mathcal{P}) = \tau(\mathcal{P}) - O(\mathcal{P})$. The **overhead ratio** of the checkpointing protocol
\( \mathcal{P} \), denoted \( v(k, \mathcal{P}) \), is

\[
v(k, \mathcal{P}) = \frac{\Gamma_k(\mathcal{P}) - \tau'(\mathcal{P})}{\tau'(\mathcal{P})} = \frac{\Gamma_k(\mathcal{P})}{\tau'(\mathcal{P})} - 1 = \frac{\Gamma_k(\mathcal{P})(F(\mathcal{P}) + 1)}{T - O(\mathcal{P})(F(\mathcal{P}) + 1)} - 1
\]

When we do not care about the checkpointing protocol, we denote the overhead ratio as \( v(k) = \frac{\Gamma_k(F+1)}{T-O(F+1)} - 1 \). Notice that \( v(k) \geq 0 \) for every \( k \geq 0 \). Moreover, a smaller overhead ratio corresponds to a better execution. Actually, the overhead ratio is the ratio between the total overhead of checkpoint/recovery and the computation events in a checkpoint interval. Therefore, for \( v(k) \geq 1 \), we have that the total overhead of checkpoint/recovery is bigger than the computation events in a checkpoint interval.

Recall that \( T \) is a constant and \( F \) can be computed by either theoretical analysis or experimental work \([18]\). Therefore, to compute the overhead ratio for a given checkpointing protocol \( \mathcal{P} \), we need to compute \( \Gamma_k(\mathcal{P}) \), which depends on the \( k \)-rollback class that the execution belongs to.

### 7.3.1 Computing \( \Gamma_k \) Using Markov Chains

We compute \( \Gamma_k \) by constructing a finite-state Markov chain \([95]\) for the \( k \)-rollback class of executions. For simplicity, we assume that \( IL(E) = 1 \) for an execution \( E \). Then, we use known techniques to extract \( \Gamma_k \) from the Markov chain. The Markov chain has a unique start state and a unique sink state. The number of states in the Markov chain depends on the \( k \)-rollback class that the execution belongs to.

#### Computing \( \Gamma_0 \)

Consider process \( p \) running in \( I_{p,i+1} \) of a 0-rollback execution, as depicted in Figure 7.2. \( \Gamma_0 \) can be computed using the 3-state Markov chain presented in Figure 7.3 \([98, 110]\). Process \( p \) starts the interval with the initial state \( i \) (related to the checkpoint \( C_{p,i} \)). A transition from state \( i \) to the sink state \( i+1 \) occurs if \( I_{p,i+1} \) is completed without failures. If a failure occurs during \( I_{p,i+1} \), then \( p \) recovers from \( C_{p,i} \). In this case, we have a transition from state \( i \) to state \( R_i \). After state \( R_i \) is entered, a transition is made to state \( i + 1 \) if no further failure occurs in \( I_{p,i+1} \) after a recovery. Otherwise, a transition is made from state \( R_i \) to itself. Notice here that the case of 0-rollback execution, where the latest checkpoint is exploited, is equivalent to Vaidya’s work presented in \([98]\).
Figure 7.3: A Markov Chain represents $I_{p,i+1}$ execution in the 0-rollback class.

Let $M$ be a Markov chain that represents the interval $I_{p,i+1}$ in a $k$-rollback execution. Let $s, t$ be states in $M$ such that there is a transition from $s$ to $t$ in $M$. We denote by $P_{s,t}$ the probability of the transition from $s$ to $t$ and by $W_{s,t}$ the expected execution time spent in state $s$ before moving to state $t$. State $s \in M$ is called recovery if a transition to this state is due to a failure. For instance, in Figure 7.3, only state $R_i$ is a recovery state. Furthermore, for each state $s \in M$, we define the variable $X_s$ to be the expected cost of reaching the sink state $i+1$ from state $s$. In fact, $X_s$ equals the expected cost of all possible paths in $M$ from state $s$ to state $i+1$ weighted by their probabilities. Therefore, the expected execution time of $I_{p,i+1}$ ($\Gamma_k$) is $X_i$.

In Figure 7.3, the value of $X_i$ equals the expected cost of going from state $i$ to state $i+1$. Since there are two possible paths from state $i$ to state $i+1$, then $X_i = P_{i,i+1}W_{i,i+1} + P_{i,R_i}(W_{i,R_i} + X_{R_i})$. Therefore, we have the following two linear equations.

\[
X_i = P_{i,i+1}W_{i,i+1} + P_{i,R_i}(W_{i,R_i} + X_{R_i})
\]

\[
X_{R_i} = P_{R_i,R_i}(W_{R_i,R_i} + X_{R_i}) + P_{R_i,i+1}W_{R_i,i+1}
\]

After solving these two linear equations and substituting $P_{R_i,i+1} = 1 - P_{R_i,R_i}$, we have that $\Gamma_0 = X_i = P_{i,R_i}\left(W_{i,R_i} + \frac{P_{R_i,R_i}}{1-P_{R_i,R_i}}W_{R_i,R_i} + W_{R_i,i+1}\right) + P_{i,i+1}W_{i,i+1}$.

### Computing $\Gamma_1$

Given an execution $E \in 1$-rollback, if a failure occurs in process $p$ during $I_{p,i+1}$, then $p$ can rollback to either $C_{p,i}$ or $C_{p,i-1}$ [13]. Therefore, in the Markov chain for $\Gamma_1$ presented in Figure 7.4, we add a new state $R_{i-1}$ to represent the possibility of recovering from $C_{p,i-1}$.

A transition to state $R_i$ represents that a recovery is made from $C_{p,i}$. Since we consider a recovery mechanism that finds the most recent recovery line, after entering state $R_i$ if there is an additional failure, then the recovery will be made from $C_{p,i}$. On the other hand, a transition to state $R_{i-1}$ represents that a recovery is made from $C_{p,i-1}$. In this case, process $p$ is rolled back to re-execute the checkpoint interval $I_{p,i}$. After state $R_{i-1}$ is entered, a transition is made to state $i$ after completing $I_{p,i}$. When a further failure occurs, a transition is made from state $R_{i-1}$ to itself.

As in the case of 0-rollback, $\Gamma_1$ equals to $X_i$ which is the expected cost from the starting state $i$ to the sink state $i+1$ in the Markov chain of Figure 7.4. We compute the variable
Figure 7.4: A Markov Chain represents $I_{p,i+1}$ execution in 1-rollback class.

Given an execution $E \in 2$-rollback, if a failure occurs in process $p$ during $I_{p,i+1}$, then the rollback propagation might go as far back as $C_{p,i-2}$. Therefore, the Markov chain of the 2-rollback class for computing $\Gamma_2$ should contain transitions from state $i$ to the recovery states $R_i$, $R_{i-1}$, and $R_{i-2}$. Such a Markov chain is presented in Figure 7.5.

The states $i$ and $i_1$ in Figure 7.5 represent the same state of the execution, where process $p$ runs in $I_{p,i+1}$. However, we use two different states in the Markov chain to capture the situation that if process $p$ has rolled back to $C_{p,i-1}$, then it will not rollback to $C_{p,i-2}$. This means that there is no transition path in Figure 7.5 from state $R_{i-1}$ to state $R_{i-2}$.
Therefore, the probabilities and costs from states \( i \) and \( i_1 \) to state \( i+1 \) are identical. Thus, \( P_{i,i+1} = P_{i_1,i+1} \) and \( W_{i,i+1} = W_{i_1,i+1} \).

As in the previous cases, the expected execution time of the checkpoint interval equals to \( \Gamma_2 = X_i \). Therefore, to compute \( \Gamma_2 \), we first build the corresponding linear system \( Ax = b \) and then solve it to extract \( X_i \). From Figure 7.5, we have the following linear equations.

\[
\begin{align*}
X_i &= P_{i,i+1}W_{i,i+1} + P_{i,R_i}(W_{i,R_i} + X_{R_i}) + P_{i,R_{i-1}}(W_{i,R_{i-1}} + X_{R_{i-1}}) + P_{i,R_{i-2}}(W_{i,R_{i-2}} + X_{R_{i-2}}) \\
X_{i-1} &= P_{i-1,i}(W_{i-1,i} + X_{i}) + P_{i-1,R_{i-1}}(W_{i-1,R_{i-1}} + X_{R_{i-1}}) + P_{i-1,R_{i-2}}(W_{i-1,R_{i-2}} + X_{R_{i-2}}) \\
X_{i_1} &= P_{i_1,i+1}W_{i_1,i+1} + P_{i_1,R_{i_1}}(W_{i_1,R_{i_1}} + X_{R_{i_1}}) + P_{i_1,R_{i-1}}(W_{i_1,R_{i-1}} + X_{R_{i-1}}) \\
X_{R_i} &= P_{R_i,R_i}(W_{R_i,R_i} + X_{R_i}) + P_{R_i,i+1}W_{R_i,i+1} \\
X_{R_{i-1}} &= P_{R_{i-1},R_{i-1}}(W_{R_{i-1},R_{i-1}} + X_{R_{i-1}}) + P_{R_{i-1},i}(W_{R_{i-1},i} + X_{i_1}) \\
X_{R_{i-2}} &= P_{R_{i-2},R_{i-2}}(W_{R_{i-2},R_{i-2}} + X_{R_{i-2}}) + P_{R_{i-2},i-1}(W_{R_{i-2},i-1} + X_{i-1})
\end{align*}
\]

After reducing the parentheses and arranging the variables on the left side of each equation, we can represent these equations by the linear system \( Ax = b \). Notice that in \( A \) the first row and column include the coefficients of equations corresponding to the non-recovery states, and the rest include the coefficients of recovery states.

\[
A = \begin{bmatrix}
1 & 0 & 0 & -P_{i,R_i} & -P_{i,R_{i-1}} & -P_{i,R_{i-2}} \\
-P_{i-1,i} & 1 & 0 & 0 & -P_{i-1,R_{i-1}} & -P_{i-1,R_{i-2}} \\
0 & 0 & 1 & -P_{i,R_i} & -P_{i,R_{i-1}} & 0 \\
0 & 0 & 0 & P_{R_i,i+1} & 0 & 0 \\
0 & 0 & -P_{R_{i-1},i} & 0 & 0 & P_{R_{i-1},i} \\
0 & 0 & -P_{R_{i-2},i-1} & 0 & 0 & P_{R_{i-2},i-1}
\end{bmatrix}
\]

\[
x = \begin{bmatrix}
X_i \\
X_{i-1} \\
X_{i_1} \\
X_{R_i} \\
X_{R_{i-1}} \\
X_{R_{i-2}}
\end{bmatrix}, \quad b = \begin{bmatrix}
P_{i,i+1}W_{i,i+1} + P_{i,R_i}W_{i,R_i} + P_{i,R_{i-1}}W_{i,R_{i-1}} + P_{i,R_{i-2}}W_{i,R_{i-2}} \\
P_{i-1,i}W_{i-1,i} + P_{i-1,R_{i-1}}W_{i-1,R_{i-1}} + P_{i-1,R_{i-2}}W_{i-1,R_{i-2}} \\
P_{i_1,i+1}W_{i_1,i+1} + P_{i_1,R_{i_1}}W_{i_1,R_{i_1}} + P_{i_1,R_{i-1}}W_{i_1,R_{i-1}} \\
P_{R_i,R_i}W_{R_i,R_i} + P_{R_i,i+1}W_{R_i,i+1} \\
P_{R_{i-1},R_{i-1}}W_{R_{i-1},R_{i-1}} + P_{R_{i-1},i}W_{R_{i-1},i} \\
P_{R_{i-2},R_{i-2}}W_{R_{i-2},R_{i-2}} + P_{R_{i-2},i-1}W_{R_{i-2},i-1}
\end{bmatrix}
\]

**Computing \( \Gamma_3 \)**

Figure 7.6 presents the Markov chain representing \( I_{p,i+1} \) in 3-rollback execution. Notice here that this Markov chain has 10 states apart from the sink state \( i+1 \). Therefore, for computing
\( \Gamma_3 \), we have to solve 10 linear equations with 10 variables, where \( \Gamma_3 = X_i \).

Figure 7.6: A Markov Chain represents \( I_{p,i+1} \) execution in 3-rollback class.

The matrix \( A \) of the linear system \( Ax = b \) is of size \( 10 \times 10 \) and the vectors \( x \) and \( b \) have 10 entries. Recall that, each row and column of \( A \) contains the coefficients of the cost variables corresponding to the states of the Markov chain of Figure 7.6 in the following order: \( X_i, X_{i-1}, X_{i-2}, X_{i+1}, X_{i+1}, X_{R_i}, X_{R_i-1}, X_{R_i-2}, X_{R_i-3} \). Notice that the states \( i, i_1 \) and \( i_2 \) represent the same state of executing \( I_{p,i+1} \) and the states \( i - 1 \) and \( (i - 1) \) represent also the same state of executing \( I_{p,i} \). As in the previous cases, we determine the linear equations \( X_s \) for every state \( s \) such that \( s \neq i + 1 \). Therefore, the matrix \( A \) equals

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-P_{i-1,i} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -P_{i-2,i} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -P_{i-1,i} & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -P_{i-1,i} & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -P_{i-1,i} & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -P_{i-1,i} & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -P_{i-1,i} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
x = \begin{bmatrix} X_i \\ X_{i-1} \\ X_{i-2} \\ X_{i+1} \\ X_{i+1} \\ X_{R_i} \\ X_{R_i-1} \\ X_{R_i-2} \\ X_{R_i-3} \end{bmatrix}, \quad b = \begin{bmatrix} P_{i,i+1} W_{i,i+1} + P_{i,R_i} W_{i,R_i} + P_{i,R_i-1} W_{i,R_i-1} + P_{i,R_i-2} W_{i,R_i-2} + P_{i,R_i-3} W_{i,R_i-3} \\ P_{i-1,i} W_{i-1,i} + P_{i-1,R_i-1} W_{i-1,R_i-1} + P_{i-1,R_i-2} W_{i-1,R_i-2} + P_{i-1,R_i-3} W_{i-1,R_i-3} \\ P_{i-2,i} W_{i-2,i} + P_{i-2,R_i-2} W_{i-2,R_i-2} + P_{i-2,R_i-3} W_{i-2,R_i-3} \\ P_{i+1,i} W_{i+1,i} + P_{i+1,R_i} W_{i+1,R_i} + P_{i+1,R_i-1} W_{i+1,R_i-1} + P_{i+1,R_i-2} W_{i+1,R_i-2} \\ P_{i+1,R_i} W_{i+1,R_i} + P_{i+1,R_i-1} W_{i+1,R_i-1} + P_{i+1,R_i-2} W_{i+1,R_i-2} \\ P_{i+1,R_i-1} W_{i+1,R_i-1} + P_{i+1,R_i-2} W_{i+1,R_i-2} + P_{i+1,R_i-3} W_{i+1,R_i-3} \\ P_{i+1,R_i-2} W_{i+1,R_i-2} + P_{i+1,R_i-3} W_{i+1,R_i-3} + P_{i+1,R_i-4} W_{i+1,R_i-4} \\ P_{i+1,R_i-3} W_{i+1,R_i-3} + P_{i+1,R_i-4} W_{i+1,R_i-4} \\ P_{i+1,R_i-4} W_{i+1,R_i-4} \\ P_{i+1,R_i-5} W_{i+1,R_i-5} \end{bmatrix}
\]

Computing \( \Gamma_k \)

Figure 7.7 presents the Markov chain of the \( k \)-rollback class. In this chain we have \( m = \frac{(k+2)(k+1)}{2} \) states apart from the sink state \( i + 1 \). Therefore, the matrix \( A \) of the linear system \( Ax = b \) is of size \( m \times m \) and the vectors \( x \) and \( b \) have \( m \) entries.

77
Figure 7.7: A Markov Chain represents $I_{p,i+1}$ execution in $k$-rollback class.

Let $\mathcal{M}$ be a Markov chain that represents the execution of $I_{p,i+1}$ in the $k$-rollback class. Consider the matrix $A$ of the linear system $Ax = b$ corresponding to $\mathcal{M}$. $A$ contains the coefficient values of the variables $X_s$ for every state $s \in \mathcal{M}$ apart from the state $i+1$. As we mentioned before, the first row and column of $A$ include the coefficient values corresponding to the non-recovery states and the following rows and columns include values of the recovery states. More precisely, the coefficient values in $A$ are arranged according to the following order of the vector, $x = \langle X_i, X_{i-1}, \ldots, X_{i-k+1}, X_i, X_{i-1}, \ldots, X_{R_i}, X_{R_{i-1}}, \ldots, X_{R_{i-k}} \rangle$

Given two states $s, t \in \mathcal{M}$ such that $s, t \neq i + 1$. The entry $A(i)(j)$ - the value of the $i$-th row and $j$-th column of the matrix $A$ - contains the coefficient value of an expected cost variable $X_s$ in the derived linear equation of an expected cost variable $X_t$. The values of $A$ can be determined from $\mathcal{M}$ by the following rules:

- If there is no transition from state $s$ to state $t$ in $\mathcal{M}$, then $A(i)(j) = 0$.
- If $s \neq t$ and there is a transition from state $s$ to state $t$ in $\mathcal{M}$, then $A(i)(j) = -P_{s,t}$.
- For $s = t$, if state $s = R_{m-l}$ is a recovery state, then $A(i)(i) = P_{R_{m-l},m-l+1}$. Otherwise, $A(i)(i) = 1$.

7.3.2 Computing the Overhead Ratio

Recall that $v(k) = \frac{\Gamma_k(F+1)}{T - \Omega(F+1)} - 1$. As we mentioned before, $T$ is constant and the expected number of forced checkpoints $F$ can be obtained either by experimental work or theoretical analysis [44]. Thus, to compute $v(k)$ we first need to compute $\Gamma_k$. Given the discussion in Section 7.3.1, we must first determine $P_{s,t}$ and $W_{s,t}$ for any pair of connected states $s$ and $t$ in the corresponding Markov chain $\mathcal{M}$.

As mentioned before, a transition from state $i$ to state $i + 1$ occurs if $I_{p,i+1}$ is completed without a failure. If this transition is made, then $\tau$ units of time are spent going from state $i$ to state $i + 1$ where $\tau = \frac{T}{F+1}$. Namely, $W_{i,i+1} = \tau$. We assumed that failures are governed by a Poisson process with rate $\lambda$. Therefore, the probability that there is no failure during $\tau$ units of time is $P_{i,i+1} = e^{-\lambda \tau}$. Since for every $l$, $0 \leq l \leq k - 1$, the probability and expected
cost from state $i - l$ to state $i - l + 1$ is equal, we have that

$$
P_{i-l,i-l+1} = e^{-\lambda \tau}$$
$$W_{i-l,i-l+1} = \tau, \quad \text{where } 0 \leq l \leq k - 1$$

(7.1)

On the other hand, if a failure occurs during the checkpoint interval $I_{p,i+1}$, then the rollback is made to the newest exploited checkpoint $C_{p,i-l}$ for $0 \leq l \leq k$. Therefore, a transition is made from state $i$ to one of the states $\{R_{i-l} \mid 0 \leq l \leq k\}$. The probability of such a transition is equal to $1 - P_{i,i+1}$. We denote $P_{i,R_{i-l}} = \mu_l(1 - P_{i,i+1})$ for $0 \leq l \leq k$ and $\sum_{l=0}^{k} \mu_l = 1$, where the value of $\mu_l$ depends on the communication pattern and the checkpointing protocol. The cost of this transition, $W_{i,R_{i-l}}$, is the expected execution time for $I_{p,i+1}$ until a failure occurs. Given that a failure occurs in the interval $[0, \tau)$ during the execution of $I_{p,i+1}$, the *time to failure* (TTF) is a random variable $x$ in the interval $[0, \tau)$ [95]. Moreover, its probability density function (PDF) is $\lambda e^{-\lambda x}$ for $0 \leq x < \tau$, and its conditional density function is

$$f_l(x) = \begin{cases} 
\frac{\lambda e^{-\lambda x}}{P_{i,R_{i-l}}} & 0 \leq x < \tau \\
0 & \text{elsewhere}
\end{cases} \quad \text{where } 0 \leq l \leq k$$

Implying that $W_{i,R_{i-l}} = \int_{0}^{\tau} x \cdot f_l(x) dx = \frac{1}{\lambda} - \frac{\tau e^{-\lambda \tau}}{1 - e^{-\lambda \tau}}$, where $0 \leq l \leq k$. Therefore, for every $m$, $0 \leq m \leq k - 1$, the probability of transition from every state $i - m$ to state $R_{i-m-l}$, $P_{i-m,R_{i-m-l}} = \mu_l[1 - e^{-\lambda \tau}]$ and $\sum_{l=0}^{k} \mu_l = 1$

$$W_{i-m,R_{i-m-l}} = \frac{1}{\lambda} - \frac{\tau e^{-\lambda \tau}}{1 - e^{-\lambda \tau}} \quad \text{where } 0 \leq l \leq k - m$$

(7.2)

For every $l$, $0 \leq l \leq k$, after state $R_{i-l}$ of Figure 7.7 is entered, a transition to state $i + 1 - l$ is made if no further failure occurs before $I_{p,i+1-l}$ is completed. As mentioned before, the execution time required to reach state $i + 1 - l$ is $W_{R_{i-l},i+1-l} = \tau + r + L - o$. Therefore, the probability that no additional failure occurs is $P_{R_{i-l},i+1-l} = e^{-\lambda(\tau + r + L - o)}$. It follows that

$$P_{R_{i-l},i+1-l} = e^{-\lambda(\tau + r + L - o)}$$
$$W_{R_{i-l},i+1-l} = \tau + r + L - o \quad \text{where } 0 \leq l \leq k$$

(7.3)

If another failure occurs after being in state $R_{i-l}$, $0 \leq l \leq k$, a transition is made from state $R_{i-l}$ to itself. For this transition we have $P_{R_{i-l},R_{i-l}} = 1 - P_{R_{i-l},i+1-l} = 1 - e^{-\lambda(\tau + r + L - o)}$. The cost of this transition, $W_{R_{i-l},R_{i-l}}$, is the expected time spent during $I_{p,i+1-l}$ until another failure occurs. As discussed above, the cost can be obtained in a similar way to $W_{i,R_{i}}$. The PDF of the TTF is $\lambda e^{-\lambda x}$ for $x \in [0, \tau + r + L - o)$. Thus we have,

$$P_{R_{i-l},R_{i-l}} = 1 - e^{-\lambda(\tau + r + L - o)}$$
$$W_{R_{i-l},R_{i-l}} = \frac{1}{\lambda} - \frac{(\tau + r + L - o)e^{-\lambda(\tau + r + L - o)}}{1 - e^{-\lambda(\tau + r + L - o)}} \quad \text{where } 0 \leq l \leq k$$

(7.4)

From Equations (7.1)-(7.4), we can extract the values of the matrix $A$ and the vector $b$ in the linear system $Ax = b$. Then, by using any known technique for solving a linear system, we can obtain the expected cost variable $X_i$. Lastly, by substituting $\Gamma_k$ in the equation of $v(k)$ we can obtain the value of overhead ratio.
7.3.3 Numerical Results

To compute the overhead ratio \( v(k) \) of an execution \( E \in k\)-rollback, we first need to set the following parameters: \( T, F, O, r, L, o, M, \) and \( \lambda \). \( T \) is the only parameter that can be specified by the user, while the other parameters are obtained from the execution \( E \) and the environment where \( E \) is executed.

Since \( T \) is specified by the user, we would like to use the optimum value of \( T \) that will yield a minimum overhead ratio. A straightforward way to compute the optimum \( T \) is by solving the equation \( \frac{\partial \gamma(k)}{\partial T} = 0 \). However, computing \( \frac{\partial \gamma(k)}{\partial T} \) is difficult. For instance, the simplest equation for \( \gamma \) looks as follows. Let \( \tau = \frac{T}{F+1} \) and \( W = \tau + r + L - o \), then
\[
\Gamma_0 = \tau e^{-\lambda r} + \left(1 - e^{-\lambda r}\right) \left[\frac{1}{\lambda} - \frac{\tau e^{-\lambda r}}{1 - e^{-\lambda r}} + \frac{1 - e^{-\lambda W}}{e^{-\lambda W} - 1} \right] + W.
\]

Therefore, in this section we present numerical results to determine the optimum \( T \) for a given set of parameter values. In addition, we analyze the impact of the rollback propagation \((k)\) on the value of the overhead ratio.

We report results from two computing environments: SMALL and BIG. SMALL represents small and short-running applications executed in the Starfish system [7]. In previous experimental work, presented in Chapter 4 and in [16], we ran a matrix multiplication application in Starfish with checkpointing. From this experimental work we obtained the following parameters: \( o = 1.78 \) seconds, \( L = 4.292 \) seconds, and \( r = 3.32 \) seconds. On the other hand, BIG represents a long-running distributed application. In this environment, applications run several days and produce large checkpoint files. For this environment, we use the simulation and experimental work performed by Plank and Thomason [84] where they obtained the following parameters: \( o = 316.7 \) seconds, \( L = 5375.5 \) seconds, and \( r = 5375.5 \) seconds. Moreover, both environments use a Fast Ethernet network (100 Mb/s). Hence, \( w_m = 0.000276 \) and \( w_b = 10^{-8} \) seconds.

In both environments we assume that the failure rate of a single process is \( \lambda = 1.23 \cdot 10^{-6} \); this value was used also in [84, 98]. In addition, at this point, we do not have enough statistical data to determine the right rollback probabilities for \( k \)-rollback executions. Instead, we investigate two options. For \( k \)-rollback execution \( E \) and process \( p \in E \), we first assume equal probabilities of recovering from the possible checkpoints. That means the probability to rollback to any \( C_{p,i-j}, 1 \leq j < k \), is the same. We also consider decreasing probabilities of recovering from the possible checkpoints. That is, if process \( p \) fails during the checkpoint interval \( I_{p,i+1} \), then the probability of recovering \( p \) from \( C_{p,j} \) is twice as big as the probability of recovering from \( C_{p,j-1} \) for \( i - k < j \leq i \) (given that \( i \geq k \)).

Figure 7.8 depicts the overhead ratio versus the length of the independent checkpoint interval \((T)\) in the SMALL and BIG environments respectively. In this figure we show the relation between \( T \) and the overhead ratio. Therefore, we set a constant value for \( F = 0, n = 8, \) and \( M = w_m + w_b \) (we discuss the impact of this choice of values later). The figure shows that for every value of \( k \) we have an optimum \( T \) that achieves the minimum \( v(k) \) in both environments. Notice that since the BIG environment has long-running applications, the optimum \( T \) values as presented in Figure 7.8(b) are larger than in the SMALL environment as presented in Figure 7.8(a). Moreover, in both environments we figure out that \( v(k) \) is a monotonic function with \( k \), and for larger \( k \) the minimum value for \( v(k) \) is obtained when \( T \) is small. This is because with larger \( k \) the execution looses less computation events with
smaller $T$.

On the other hand, Figure 7.9 depicts the overhead ratio versus $T$ as in Figure 7.8(a) but with decreasing probabilities of recovering from the possible checkpoints. We also have here an optimum $T$ for every value of $k$. However, $v(k)$ grows much more slowly with $k$, and as can be seen, $v(16)$ is almost the same as $v(25)$.

Consider Figure 7.8(a) and Figure 7.9(a). For every $k$, $v(k)$ obtains the minimum value in the interval $[0, 500]$ of $T$. In this interval the value of $v(k)$ drops exponentially while $T$ increases. This is because in the SMALL environment $o$ and $L$ are small, so we need longer checkpoint interval for performing useful computation and thus achieving better overhead ratio. However, since with large values of $T$ we can lose computation events due to rollback, in the interval $(500, \infty)$ the overhead ratio becomes much worse as long as $T$ becomes longer (particularly in Figure 7.8(a) due to equal probabilities). Therefore, there is a value of $T$, where the lose of overhead ratio due to the checkpoint overhead and latency on one hand and the lose due to rollback propagation on the other hand are similar. This value of $T$ yields the minimal overhead ratio.

Figure 7.9(b) shows the tradeoff between the optimum value of $T$ and $\lambda$. With no doubt, the overhead ratio increases as the failure rate increases too. Thus, to limit the losing of useful computation with large values of $\lambda$, we need to set smaller values of $T$.

We next investigate the effect of the values of $F$ and $n$ on $v(k)$. Figure 7.10(a) shows that the optimum value of $T$ is obtained in different points in respect with $F$. Figure 7.10(a) shows that the optimum value of $T$ is obtained in different points in respect with $F$. Notice that in Figure 7.10(a) the overhead ratio with $F = 2$ and $T = 3000$ is equivalent to the overhead ratio with $F = 0$ and $T = 1000$. In general, the overhead ratio with $F = f$ and $T = t$ is equivalent to the overhead ratio with $F = 0$ and $T = \frac{t}{t+1}$. Therefore, there is no reason to present the overhead ratio with $F > 2$. On the other hand, Figure 7.10(b)
shows that the overhead ratio increases proportionally with number of processes. Under the assumption that process failures occur randomly and independently with probability of $p$, the probability of a failure in a system with $n$ processes is $1 - (1 - p)^n$. Therefore, the overhead ratio increases proportionally with $n$.

Figure 7.10: The overhead ratio is affected by other parameters like $F$ and $n$.

### 7.4 Performance Analysis of Checkpointing Protocols

There are several checkpointing protocols in the literature [44]. In this section, we present several distributed checkpointing protocols and analyze the performance of those protocols
by using the overhead ratio measure.

7.4.1 Distributed Checkpointing Protocols

In this section we present various protocols from different $k$-rollback classes. We briefly describe each protocol $\mathcal{P}$, show the $k$-rollback class to which it belongs, and discuss $F(\mathcal{P})$ and $M(\mathcal{P})$.

Checkpoint-After-Send-Before-Receive [104]

In the Checkpoint-After-Send-Before-Receive (CASBR) protocol, a checkpoint must be taken after every send event and before every receive event. Clearly, every global checkpoint is a recovery line. Thus, CASBR $\in 0$-rollback.

In this protocol there is a forced checkpoint with every send/receive event, namely, $F(\text{CASBR}) = 2 \cdot MR(E)$ for a given execution $E$. Notice that there are no control messages in CASBR. Therefore, $M(\text{CASBR}) = 0$.

Sync-and-Stop [80]

Sync-and-Stop (SaS) is a coordinated checkpointing protocol where a coordinator creates a synchronization barrier among all the processes and then all the processes take a global checkpoint which is also a recovery line. Specifically, the protocol works as follows:

1. The coordinator stops its target program and broadcasts a special message, $\text{chkpt.request}$, to all other processes.

2. Upon receiving the $\text{chkpt.request}$, each process stops running its target program and sends the message $\text{chkpt.ready}$ to the coordinator.

3. When the coordinator collects $\text{chkpt.ready}$ messages from all other processes, it broadcasts the message $\text{chkpt.do}$ and takes a local checkpoint.

4. When a process $p$ receives the $\text{chkpt.do}$ message, it takes a local checkpoint and then sends back $\text{chkpt.done}$.

5. When the coordinator receives $\text{chkpt.done}$ from all other processes, it broadcasts the message $\text{chkpt.commit}$ to resume the application execution.

Obviously, the collection of checkpoints is a recovery line. Thus, the checkpointing protocol SaS $\in 1$-rollback [13]. In this protocol there are no forced checkpoints. In each process the checkpoint interval equals to $T$, therefore, $F(\text{SaS}) = 0$. Regarding the control overhead, in each phase of SaS, the coordinator broadcasts three messages and the other $n-1$ processes send two reply messages. Moreover, notice that the protocol works correctly with an 8-bit control messages. Therefore, $M(\text{SaS}) = 5(n-1)(w_m + 8 \cdot w_b)$. 

83
Chandy-Lamport [34]
The Chandy-Lamport (C-L) protocol is a coordinated checkpointing protocol in which there is no need to block the application execution. In C-L the coordinator initiates a checkpoint by broadcasting marker messages and then takes a local checkpoint. When another process, say \( p \), receives a marker message from a channel \( c \), it takes the following steps:

- If \( p \) has not taken a checkpoint, then it broadcasts the marker, takes a local checkpoint, and records the state of \( c \) as being empty.
- Otherwise, \( p \) records the state of the channel \( c \) as the sequence of messages received along \( c \) after it had taken a checkpoint and until receiving the marker.

C-L belongs to 1-rollback and since there are no forced checkpoints, \( F(C-L) = 0 \). Regarding control messages, in a fully connected network with \( n \) nodes, C-L generates \( 2n(n-1) \) messages per checkpoint [80] and the marker takes 8 bits where it should distinguish between different runs of C-L. Therefore, \( M(C-L) = 2n(n-1)(w_m + 8w_b) \).

Fixed-Dependency-Interval [104]
In the Fixed-Dependency-Interval (FDI) protocol, when a process \( p \) sends an application message, it piggybacks its dependency vector \( D_p \). On the other hand, when a process \( q \) receives a message \( M \), it performs the following:

- If \( M.D[p] \leq D_q[p] \) for all \( p, 1 \leq p \leq n \), then deliver \( M \).
- Otherwise, process \( q \) takes a forced checkpoint and updates \( D_q \) accordingly and then delivers \( M \).

Wang showed that FDI is Z-path free (ZPF) [104]. From Chapter 6, we know that ZPF \( \subseteq 1 \)-rollback. Therefore, FDI \( \in 1 \)-rollback. Also, the dependency vector is piggybacked on each message. Thus, \( M(FDI) = n \cdot MR(E) \) for an execution \( E \). However, the number of forced checkpoints clearly depends on the communication pattern. To guarantee ZPF, this protocol takes many forced checkpoints that depend on the communication pattern. We assume that \( F(FDI) = c \cdot MR(E) \) for \( 0 \leq c \leq 1 \). In the measurements below, \( c = 0.1 \).

Briatico, Ciuffoletti and Simoccini (BCS) [29]
The BCS protocol is a communication induced checkpointing protocol. In this protocol, each process \( p \) maintains a checkpoint timestamp \( lc_p \) and acts as follows:

1. Whenever \( p \) takes a checkpoint, it increases \( lc_p \) by 1.
2. For each message \( m \), \( p \) piggybacks \( lc_p \) on \( m \). We denote the piggybacked value as \( m.lc \).
3. Whenever \( p \) receives a message \( m \), it compares \( lc_p \) with \( m.lc \). If \( m.lc > lc_p \), then \( p \) sets \( lc_p \) to \( m.lc \) and takes a forced checkpoint before delivering \( m \).
Manivannan and Singhal [73] proved that BCS belongs to the ZCF class, and from Chapter 6 we know that $ZCF \subset n$-rollback. Regarding forced checkpoints, it can be claimed that the number of forced checkpoints depends on the communication pattern of an execution $E$. Running different applications under BCS, Alvisi et al [18] showed that the expected number of forced checkpoints in BCS is 2 for $T = 360$ and $T = 480$ seconds. Lastly, since BCS piggybacks a 32-bit logical clock (assuming a 32-bit architecture) on every application message, then $M = MR(E)(32 \cdot w_b + \epsilon)$, where $\epsilon$ is the delay time due to intercepting every data message. For instance, in our measurements presented later, $\epsilon = 50$ microseconds.

**Baldoni, Quaglia and Ciciani (BQC) [22]**

The BQC protocol is another communication induced checkpointing protocol. BQC ensures ZCF by preventing potential Z-cycles from being created. Since this protocol is in ZCF, then by Chapter 6, BQC $\in n$-rollback. Regarding forced checkpoints, Alvisi et al [18] showed that BQC is worse than BCS but $F(BQC) = 2$. Lastly, the protocol propagates $n^2$ 32-bit values on each application message to help processes detect suspected Z-cycles. Therefore, $M(BQC) = MR(E)(32 \cdot n^2 \cdot w_b + \epsilon)$, where $\epsilon$ is the delay time due to intercepting every data message.

**$d$-Bounded Cycles [13]**

In Section 6.4, we present the $d$-BC checkpointing protocol. By Theorem 6.3.3, $d$-BC belongs to $(n-1)d$-rollback. Upon a new checkpoint $C_{p,i}$, process $p$ broadcasts a cut of size no more than $d \cdot n$. Therefore, $M(d-BC) = n \cdot w_m + d \cdot n^2 \cdot w_b$. Regarding forced checkpoints, $d$-BC forces checkpoints by calling C-L only if a cycle of size $d$ is generated. Since a Z-cycle is a special case of a cycle, then the conditions for generating cycles and Z-cycles are almost equivalent. Also since $ZCF = 1$-BC, then by [18], $F(1$-BC$) = 2$. Furthermore, $F(d$-BC$)$ decreases as $d$ increases.

Figure 7.11 shows that the overhead ratio increases linearly with $d$ in $d$-BC executions. This is because the rollback propagation increases proportionally with $d$.

### 7.4.2 Comparing Distributed Checkpointing Protocols

We compute the overhead ratio for the checkpointing protocols presented in the previous section. We use both the SMALL and BIG environments to compare the performance of these protocols. First we compare the coordinated checkpointing protocols: C-L and SaS. Figure 7.12 depicts the overhead ratio of these protocols in both SMALL and BIG environments. We used the SMALL environment with $T = 1024$ seconds, which is the optimum value of $T$. We can see in Figure 7.12(a) that C-L is better if the number of processes is less than 100 and it becomes worse for large number of processes. This is because C-L incurs more control overhead than SaS, where $M(C-L) = O(n^2)$ and $M(SaS) = O(n)$. In the BIG environment we have the same relation between SaS and C-L. Since in this and all following measurements,
Figure 7.11: The overhead ratio increases linearly with $d$ in $d$-BC executions.

both the BIG and SMALL environments exhibit the same qualitative behavior, we only present the results for SMALL from now on.

Figure 7.12: Comparing the SaS and C-L protocols with various number of processes.

In Figure 7.13, we compare several checkpointing protocols. We computed the overhead ratio in the SMALL environment and $T = 1024$ seconds. Consider the $n$-rollback protocols: BQC, BCS and 1-BC. The 1-BC protocol is an optimistic protocol that takes fewer forced checkpoints than BQC and BCS. Moreover, 1-BC does not use piggybacking as the other protocols do, therefore, its overhead ratio is not affected by the communication pattern. However, 1-BC incurs more control overhead than the other protocols due to the control messages it produces. As depicted in Figure 7.13(a), in executions with small MR, 1-BC has worse overhead than BCS and BQC. On the other hand, when MR is large, 1-BC performs
better than both BQC and BCS. Figure 7.13(b) shows that with $MR = 2098$, BQC and BCS are worse than in Figure 7.13(a). Notice that, since BQC piggybacks $O(n^2)$ control information while BCS piggybacks only an 8-bit logical clock, then $v(n, BCS) \leq v(n, BQC)$.

Figure 7.13: Comparing checkpointing protocols of different classes with equal probabilities for rollback.

Figure 7.14 compares the same protocols as Figure 7.13, with the same MR and $T$ values, but with decreasing probabilities of recovering from the possible checkpoints. Note that the overhead ratio in Figure 7.14 is smaller than in Figure 7.13. This is because when the probability of long rollbacks is small, the probability of loosing computation events is small as well.

Clearly, we can see in Figure 7.14 that the overhead ratio of executions with protocols that use piggybacking are worse than others for large MR. These protocols are FDI, BCS, and BQC. For presentation reasons, CASBR is not presented here since it has the worst overhead ratio. This is because CASBR is affected only by the communication pattern, where for every send/receive event there is a forced checkpoint. Notice that in Figure 7.14(b), 1-BC has better overhead ratio than the other $n$-rollback protocols. Lastly, we can conclude here that the C-L protocol is the best for up to 32 processes. Recall that SaS becomes better than C-L for large number of processes as depicted in Figure 7.12.
Figure 7.14: Comparing checkpointing protocols of different classes with decreasing probabilities for rollback.
Chapter 8
Summary and Future Work

8.1 Review

Clusters of workstations offer a potential for cost effective high-performance computing. However, building usable clusters is inherently a difficult task. Successful implementations of such clusters must retain high-performance, while addressing issues like manageability, fault-tolerance, high-availability, and coping with dynamic changes in the environment.

The main goal of this thesis is to obtain reliability in high performance distributed systems. We have presented a framework in which reliability can be provided by applying C/R protocols. We define a distributed computation model with checkpoint, recovery, and crash events. Moreover, we define basic notions of performance analysis of distributed systems that employ C/R protocols.

Following this, we have presented the design and architecture of Starfish, a distributed system that tries to provide reliability without sacrificing performance. Starfish uses a unique approach in providing reliability by combining C/R and group communication techniques. We believe that the Starfish architecture is novel in the specific way it addresses all of the above concerns. Starfish serves as a good testbed for new C/R protocols and is also a fairly portable system in its design. The performance of the initial Starfish release (currently, we have release 0.2) is promising, although some additional fine tuning is still needed.

We have described a system architecture that brings fault-tolerance and migration to scientific users who need not be computer systems experts. That is, integrating Starfish checkpoint library with NetSolve and IBP. There are two main limitations to this system. First, if a user is not making use of the core NetSolve system services (e.g. linear algebra subroutines) listed in section 4.6, then the “not an expert” label applies less forcefully, as the user must learn how to configure the NetSolve servers. Although this task is made easier by Java-based tools, it is a level of complexity higher than simply using NetSolve from Matlab or Excel. The second limitation is the restriction that the C/R machine must be of the same architecture. This limitation arises from the fact that the used Starfish library is a core-dump style checkpointer (presented in Appendix A). The architecture of the system could easily be extended to use more portable checkpointing substrates, such as applications that implement their own checkpointing and rollback recovery with the help of libraries such as
libft [58], or a toolkit that embeds portable checkpoints into arbitrary programs [86]. We are exploring using the Porch toolkit [6] to add portable checkpointing to the core NetSolve services.

In order to make Starfish a reliable heterogeneous distributed system, we implemented Starfish in a virtual machine language and provided a heterogeneous C/R mechanism that works across a wide range of hardware architectures and operating systems. This mechanism is integrated into the OCaml virtual machine. The main benefits of our approach, aside from simplifying the problem of heterogeneous C/R, is that by working at the virtual machine level we can eliminate many typical restrictions that exist in many other user level C/R approaches, as well as obtaining smaller checkpoint files.

Checkpointing is used to limit the loss of computation that may result from a failure. However, C/R incurs an overhead that decreases the system performance. Furthermore, in order to provide an efficient distributed checkpointing protocol in Starfish, we need to consider the rollback propagation, which is an important parameter in determining the effectiveness of a C/R mechanism. An aggressive protocol may take more checkpoints, and therefore add more overhead. Thus, the total number of induced checkpoints is another important criterion when examining such protocols. In this dissertation, we explored the tradeoff between the aggressiveness of a condition for forcing induced checkpoints, and the number of checkpoints that may need to be rolled back following a failure when using a protocol that implements the condition. Specifically, we have introduced the notion of \(k\)-rollback, which quantifies this number, and investigated the relation of known distributed checkpointing techniques to the corresponding \(k\)-rollback hierarchy.

Our work shows that there is a correlation between the aggressiveness of a condition for forcing induced checkpoints, and the rollback propagation. In general, with the exception of the 1-rollback class, forcing induced checkpoints in a more aggressive manner translates into shorter rollbacks. In particular, we have shown that the rollback length of ZCF protocols may be up to \(n - 1\) times longer than of ZPF protocols. Also, to show that the hierarchy is not trivial, we have presented the \(d\)-BC class, which is a generalization of ZCF, and showed that \(d\)-BC protocols belong to the class \(((n - 1) \cdot d)\)-rollback. We have also developed a parameterized novel protocol that guarantees \(d\)-BC, for any value of \(d\). This protocol has the following three appealing properties: It does not piggyback control information on data messages, the number of control messages is linear with the number of checkpoints, and each node is always aware of a very recent recovery line, allowing for efficient garbage collection of old checkpoint files.

Finally, we have presented an evaluation tool, called overhead ratio, for performance analysis of distributed executions. This tool considers all the parameters that affect the execution in evaluating distributed checkpointing protocols. By using Markov chain techniques and solving linear equations, we can compute the overhead ratio for any distributed execution. Consequently, by using the overhead ratio, we have compared several checkpointing protocols in different environments. All the results we obtained show that the two parameters that affect dramatically the values of overhead ratio are the propagation rollback and communication pattern. Therefore, in order to apply a checkpointing protocol with minimum overhead ratio, it is proper to use a protocol with small rollback propagation and which
does not piggyback messages. We found that in a couple of realistic setting, coordinated checkpointing protocols, namely C-L and SaS, performs better than induced checkpointing protocols, since they satisfy these conditions.

Moreover, as in uniprocessor systems [98], we showed that for every distributed checkpointing protocol, for every environment there is an optimum value of $T$ that achieves the best overhead ratio.

### 8.2 Future Work

#### 8.2.1 Starfish

Starfish supports fault-tolerance by applying C/R protocols. Besides fault-tolerance, C/R is useful and facilitates the implementation of the following features in distributed systems [109]: migration, load-balancing, and debugging. Migration is implemented by checkpointing the process and then restarting it on any machine of Starfish. Load balancing is implemented by using the C/R mechanism and statistical data collected from Starfish nodes. Lastly, debugging is implemented by using the checkpointed information where we can rollback and replay to any previous point in a taken execution path. Therefore, we plan to add these features in Starfish. Already, there is a student project for implementing load-balancing in Starfish. In addition, on the Starfish official web site [7] we have published a list of open projects.

The distributed C/R protocols implemented in Starfish, other than C-L, suffer from the following problem. In order to truly save a valid distributed state, they need to implement a mechanism that keeps track of intransit messages and store them as well. This is wasteful, since both TCP and the Myrinet driver [4] already keep track of these messages. Our goal is to save and restore these messages and the connection state from within the network mechanism, which would eliminate this redundancy.

A couple of years have passed since the first release of Starfish. We have yet to succeed in convincing people to use Starfish for long-running applications. It seems that we need to invest resources for continuing Starfish development and support. Fortunately, some researchers and students have used Starfish for different purposes. For instance, NetSolve [5] uses the C/R technique of Starfish. Moreover, students around the world have studied Starfish as part of their projects. As a result, we have gotten much feedback about the system.

#### 8.2.2 Heterogeneous Checkpointing

The main drawback of our approach of heterogeneous checkpointing is that it relies on bytecode execution, which is slower than native code execution. However, we consider our work as a first step toward achieving heterogeneous C/R. That is, we would like to extend our work in two immediate directions, namely, supporting native code generated from OCaml and bytecode Java and/or C#.

In considering native code, it appears that the main difficulties lie in the fact that native code is executed directly by the hardware, relying on the
native machines registers, heap, stack, etc. Thus, there is a need to translate these concepts between the machines and OSs. Moreover, since the native code generated by different compilers might be different, it might not be possible to map a given instruction on one machine to a corresponding instruction on another machine. Of course, one option is to turn off all optimizations, so it is possible to trace back machine level instructions to their corresponding source code instructions. In this approach, a safe point would be the border between machine level instructions that correspond to a single source code instruction. This solution is problematic, since unoptimized code runs much slower than optimized code, especially on new hardware architectures.

Another idea might be to declare the beginning and ending of functions and loops as safe points. The problem with loops is that many optimizers might reorder instructions from inside the loop to the outside. Even worse, in many scientific programs the entire code might reside within one loop. Thus, this restriction would prevent taking checkpoints during the entire “interesting” part of the computation. With functional programs, such as OCaml, it is common to place everything within functions, so normally a loop consists of multiple function calls. However, the OCaml optimizer can inline functions, which might mean that for pure functional programs, function boundaries might not always serve as good safe points.

Finally, even if we solve the problem of safe points, this is still not sufficient, since we would have to translate the function/procedure calling stack to reflect the new representation of the return addresses and parameters. Nonetheless, the fact that OCaml relies on its garbage collection and is a strongly typed language would greatly simplify these tasks.

### 8.2.3 Distributed Checkpointing Protocols

In this dissertation we defined the notion of $k$-rollback which quantifies the rollback propagation in distributed checkpointing. Looking to the future, it might be interesting to find specifications and protocols matching interesting missing entries in the $k$-rollback hierarchy. Examples of such important entries are, of course, $\log(n)$-rollback and $\sqrt{n}$-rollback.

In addition, an interesting question for future research direction is proving more lenient conditions for inducing checkpoints that yield fewer checkpoints. For example, it is clear that SZPF forces processes to take checkpoints in places that ZPF does not. Yet, one can envision scenarios in which taking an induced checkpoint under SZPF, when ZPF does not require it, might save taking additional induced checkpoints by ZPF later on. Finally, most protocols for ZPF and SZPF involve piggybacking control information on data messages. It is interesting to check the feasibility of obtaining these conditions without piggybacking and without sending too many control messages, as in our protocol.

### 8.2.4 Performance Analysis

In his book of “The Art of Computer Systems Performance Analysis”, Raj Jain claimed that [60],

92
Do not trust the results of a simulation model until they have been validated by analytical modeling or measurements.

Do not trust the results of an analytical model until they have been validated by a simulation model or measurements.

Do not trust the results of a measurements until they have been validated by simulation or analytical modeling.

We have presented analytical modeling for performance analysis of distributed executions with various checkpointing protocols. By Jain, in order to validate our results, we either need to build a simulation model for distributed execution or perform measurements. Indeed, we have started implementing several checkpointing protocols on Starfish for obtaining measurements. Currently, we have only implemented Stop-and-Sync, Chandy-Lamport, and $d$-BC. We still need to implement other interesting protocols (see Section 7.4) to validate our analytical results.

Our primary results on evaluating distributed checkpointing protocols show that coordinated checkpointing protocols are the most efficient protocols. In addition to their implementation simplicity, they ensure 1-rollback execution and do not incur forced checkpoints. Our conclusion is that looking for highly efficient coordinated checkpointing protocols is a more promising research direction than induced checkpointing protocols.
Bibliography


Appendix A

Local Checkpointing in Starfish

By supporting C/R, Starfish can provide fault-tolerance and then reliability. Moreover, there are several features that can be provided in Starfish using C/R (e.g., process migration, gang scheduling).

In this appendix we present our experience of implementing homogeneous local (or serial) C/R in Starfish. We implemented a serial C/R mechanism in the Linux operating system, but it works on any UNIX-like systems.

A.1 Checkpointing

In Linux, serial checkpointing involves saving the global data segment, the stack segment, registers, etc. In our implementation, we first save the data segment, then we save the data of OCaml virtual machine (in the case of bytecode running), and finally we save the stack and registers contexts.

The data of a Linux process consists of static data and the allocated dynamic data from the heap. In Linux, all of these data are continuous in the virtual memory, which is depicted in Figure A.1.

As presented in Figure A.1, in order to save all the data, we need to save the global data area and the allocated data on the heap. First we save the size of the saved data and then the data itself. Figure A.2 presents a pseudo-code of these operations.

After saving the data of a process, we save the process’s stack. We first call setjmp() in order to save registers states and then save all the stack area in the checkpoint file. These operations are presented as pseudo-code in Figure A.3.

A.2 Restarting

In restart, we use the checkpoint file in order to execute a prematurely stopping process. We need to restore data, stack, and registers. In order to restart a serial application we
first restore the global data area and the heap. We then restore registers and stack context. Actually, restoring the data is the opposite of the operations presented in Figure A.2. Furthermore, in order to restore a stack properly, we first jump to a temporary stack in the data segment, then we overwrite the saved stack to the current stack, and lastly we jump back to the restoring stack. Figure A.4 presents a pseudo-code for illustrating stack restoration.
**saveData** (int crFd) {
    begin = &data_start
    end = sbrk(0)
    size = end - begin
    writeOnChkptFile(crFd, &size, sizeof(size))
    writeOnChkptFile(crFd, begin, size)
}

Figure A.2: Save the data segment in the checkpoint file.

**saveStack** (int crFd) {
    if(setjmp(env) == 0) {
        /* Perform checkpointing */
        highAddr = 0xC0000000
        lowAddr = env → SP
        size = highAddr - lowAddr
        writeOnChkptFile(crFd, &size, sizeof(size))
        writeOnChkptFile(crFd, lowAddr, size)
        writeOnChkptFile(crFd, env, sizeof(env))
    } else {
        /* Perform Restarting */
        ...
    }
}

Figure A.3: Save the stack segment in the checkpoint file.

**restoreStack**(int crFd) {
    if(setjmp(env) == 0) {
        /* We still running on the original stack. Go to a temporary stack.. */
        To do this set the SP to point in a temporary stack area. */
        env → SP = ((tmpStack + TMP_STACK_SIZE) - SAVE_EXTRA_STACK) & 0xf
        longjmp(env, 1)
    } else {
        /* We are running on the temporary stack */
        highAddr = 0xc0000000
        readFromChkptFile(crFd, &total_size, sizeof(&total_size))
        size = total_size - sizeof(jmp_buf)
        lowAddr = highAddr - size
        readFromChkptFile(crFd, lowAddr, size)
        readFromChkptFile(crFd, env1, sizeof(env1))
        longjmp(env1, 1) /* Gogo to the original and restored stack */
    }
}

Figure A.4: Restore the stack segment from a checkpoint file.
Appendix B

Implementing $d$-BC in Starfish

In this appendix we describe the implementation of the $d$-BC checkpointing protocol in Starfish. We presented the protocol in Section 6.4, therefore, we assume that the reader is familiar with the protocol details. Moreover, we present here some implementation issues and the main code of the protocol in Starfish. As Starfish, $d$-BC is implemented in OCaml [10].

B.1 Data Structures

Listed below are the important data structures that each process $p$ maintains.

- **finishTime** - A global finish time variable used for computing finish times in the recursive depth-first search (DFS) [38] algorithm used in `findClassBorders` function (presented in Chapter 6).
- **d_bound** - Contains the value of the parameter $d$ in the $d$-BC protocol.
- **recursionDepth** - A global variable used to indicate the recursion depth of the `tryToAdvance` recursive function.
- **sccNumber** - A global variable that contains the number of different strongly connected components in the dependency graph as presented in Section 6.4.
- **classBorders** - A global variable that holds the pre- and post-borders that are obtained by calling `findClassBorders`.
- **depMatrix** - The dependency graph matrix. This is the $M_p$ matrix as presented in Section 6.4.
- **graph_of_intervals** - This data structure represents the dependency graph. Recall that the vertices of the graph represent the checkpoint intervals. It is an array of hash tables. The entries in every hash table contain the checkpoint intervals as saved in the `depMatrix` Matrix.
**transpose_graph** - This data structure contains the transpose of the graph **graph_of_intervals**. Given a graph \( G = (v, E) \), the transpose of \( G \) is the graph \( G^\top = (V, E^\top) \), where \( E^\top = \{(u, v) : (v, u) \in E\} \).

### B.2 The Protocol

As mentioned in Section 6.4.2, there are three types of events that may occur in process \( p \). The first event is receiving an application message from another process. Upon this event, process \( p \) updates its dependency graph matrix accordingly. The second event is taking a local independent checkpoint, and the third event is receiving a control message.

Figure B.1 depicts the OCaml code of the function **update_dependency** which is invoked upon receiving a data message from another process. This function is invoked in the VNI layer of Starfish, where the system can intercept any receiving message.

```ocaml
let update_dependency from_process to_process from_index to_index =
  let updated = ref false in
  let list = depMatrix.(to_process).(from_process) in
  for l = 0 to ((List.length list) -1) do
    let x = List.nth list l in
    if (x.to_int = to_index) then
      begin
        x.from_int <- from_index;
        depMatrix.(to_process).(from_process) <- list;
        updated:=true;
      end
  done;
  if (!updated = false) then begin
    let elt = { from_int = from_index; to_int =to_index } in
    depMatrix.(to_process).(from_process) <- (elt::list)
  end;

Figure B.1: Process \( p \) receives a data message from another process.
```

### B.2.1 Find Class Borders

As presented in Figure 6.7, **findClassBorders** is called from within **tryToAdvance**. In this function we try to recognize whether the checkpoint under scrutiny is internal to a cycle \( \Phi \) or not. If not, we return the checkpoint itself as a singleton set, otherwise we return \( \text{pre}(F) \) and \( \text{post}(F) \) (if exists) for a cycle class \( F \) such that \( \Phi \in F \).

For finding the pre- and post-borders of a cycle class \( F \), we construct a graph from the **depMatrix** matrix. Then we use DFS with the Decreasing Finish version [38] for computing
the strongly connected components of the dependency graph \texttt{graph of intervals}. The algorithm works as follows.

1. Call DFS on \texttt{graph of intervals} and get the finish times value \( f(u) \) for every vertex \( u \). The finish time is the time that DFS visited \( u \).

2. Compute the transpose graph \texttt{transpose graph} of \texttt{graph of intervals}. Simultaneously, store the \( f(u) \) values in ascending order in a different array.

3. Call DFS on \texttt{transpose graph} in decreasing order of finish times. As we proceed through the DFS algorithm, we save the DFS trees, the collection of which will be the Depth-First Forest.

4. The set of vertices in each tree will be a strongly connected component of \texttt{graph of intervals}. Hence, from this set we get the checkpoints involved in a cycle or a cycle class. If we get no such tree, then the checkpoint in scrutiny is not involved in a cycle and hence will be returned as a singleton set.

Figure B.2 presents the implementation of \texttt{findClassBorders} in Starfish as itemized above.

```
let findClassBorders checkpoint =
    reclaim_data(); (* Initialize data structures to 0 *)
    build_graph();
    compute_finish_times();
    build_transpose_graph();
    run_dfs_transpose();
    let scc = sort_scc checkpoint in
    let post_exist = update_borders scc checkpoint in
    (post_exist)
```

Figure B.2: The code of \texttt{findClassBorders} as implemented in Starfish.

Finally, below we present the code of \texttt{tryToAdvance}, which is called upon receiving a control message.

```
let rec tryToAdvance checkpoint =
    let rv = ref true in
    let condition = ref false in
    let break = ref false in
    let recursionDepth := !recursionDepth + 1;
    let post_exist = findClassBorders checkpoint in
    let pre_border = !class_borders.preBorder in
    let q = checkpoint.process in
```

107
let j = checkpt.index in
let preF = match pre_border with
| Cut(x) -> x
in
let cp = preF.(q) in
let cpt = match cp with
| CP(x) -> x
in
let border_index = cpt.index in
if ((j - border_index) = !d_bound) && (!recursionDepth = 1)) then begin
  (* now trying CL among part of the processes *)
  let result = activate_cl preF in
  if (result = true) then
    (* starting cl succeeded *)
    ...
  else
    (* starting cl failed *)
end;
let post_border = !class_borders.postBorder in
let postF = match post_border with
| Cut(x) -> x
in
if (post_exist) then begin
  for p = 0 to !numberOfProcesses - 1 do
    let r = rec_line.(p) in
    let chpt = { process = p; index = r } in
    let path_exist = zpath_exist chpt postF in
    if ((postF.(p) = NullCp) && (path_exist = true)) then begin
      rv:=false; (* return failure *)
      condition:= true;
    end;
  done;
end;
if (!condition = false) then begin
  let proc_list = ref [] in
  for p = 0 to !numberOfProcesses - 1 do
    let fcp = postF.(p) in
    if (fcp <> NullCp) then begin
      let rcp = match fcp with
        | CP(x) -> x
      in
      let newList = advancement rcp in
      proc_list:= !proc_list@newList;
  end;
end;
let length = List.length !proc_list in
for k = 0 to length -1 do
  let pr = List.nth !proc_list k in
  let last = last_line.(pr) in
  let recl = (rec_line.(pr) + 1) in
  break:=false;
  for l = last to recl do
    if (!break = false) then begin
      let new_cp = { process = pr; index = l } in
      let succeed = tryToAdvance(new_cp) in
      if (succeed = true) then
        break:=true;
    end;
  done;
done;
done;
done;
rv:=true;
end;
(!rv)
Appendix C

A Matlab Program for Computing the Overhead Ratio

The following Matlab program computes the value of \( v(2) \) (overhead ratio) in the SMALL environment, as presented in Section 7.3. In the initialize phase, we set the value of the parameters for computing both the probability and cost of a transmission in a Markov chain. Then, build the matrix \( A \) and the vector \( b \) of the linear system \( Ax = b \). After computing \( \Gamma_2 \), finally, we compute the value of \( v(2) \).

% The SMALL environment
% Computing v(k)

%Initialization
w_m = 0.000276; % In Fast Ethernet (IP) (in seconds)
w_b = 0.00000001; % This is the process in Fast Ethernet 100 Mb/s (in seconds)
MR = 2^10; % Message Rate
n=2; % Number of processes
k=2; % The rollback propagation
msgNum = 1; % Number of control messages
msgSize = 1; % The size of control information (in bits)
T=2^10; % The time between independent checkpoints (in seconds)
F=0; % The expected number of forced checkpoints
o=1.78; % The checkpoint overhead in the SMALL environment (in seconds)
L=4.292; % The checkpoint latency in SMALL (in seconds)
r=3.32; % The recovery overhead in SMALL (in seconds)
M = msgNum * w_m + msgSize * w_b; % The control overhead
O = o + M; % The total overhead
lambda=1.23*10^(-6) * n; % The probability of a failure
tau = T/(F+1); % The checkpoint interval
e=exp(1); % The value of e

%Computing the probabilities and costs in the Markov chain
Pii=e^(-lambda*tau);
Wii=tau;
tmp = tau+r+L-o;
PRiRi = 1 - e^((-lambda)*tmp);
WRiRi = lambda^(-1) - (tmp * e^((-lambda)*tmp) / PRiRi);

PRii = 1 - PRiRi;
WRii = tmp;

PiRi = (k+1)^(-1) * (1 - e^(-lambda * tau));
WiRi = lambda^(-1) - (tau * Pii)/(1 - e^(-lambda * tau));

% Compute the vector b of size 1xm, where m=(k+2)(k+1)/2
m = (k+2)*(k+1)/2;
index=0; for i=1:k
  for j=(k-i+1):-1:1
    index=index+1;
    b2(index) = Pii*Wii + (j+1)*PiRi*WiRi;
  end;
end; tm = (k*(k+1)/2); for i=tm+1:m
  b2(i) = PRiRi*WRiRi + PRii*WRii;
end;

% Computing the matrix A of size mxm, where m=(k+2)(k+1)/2
for i=1:m
  for j=1:m
    A2(i,j) = 0;
    if (i==j)
      if (i > tm)
        A2(i,j)=PRii;
      else
        A2(i,j) = 1;
      end;
    end;
  end;
end;

in=0; for j=k:-1:1
  for i =0:j-1
    in = in+1;
    if (in > tm)
      break;
    end;
    for i1 = i+1:(k+1)-(k-j)
      A2(in,tm+i1) = -PiRi;
    end;
  end;
jmp=0; for i=k:-1:1
    for j=2:i
        index = jmp + j;
        A2(index,index-1) = -Pii;
    end;
    jmp = jmp + i;
end;

inc1=0; inc2=0; for i=tm+2:m
    inc2 = inc2 + inc1;
    A2(i,tm-inc2)= -PRii;
    inc1 = inc1+1;
end;

% Compute Gamma_2
x2 = A2\(b2'); % Solving the linear system Ax=b
Gamma_2 = x2(1);
v_2 = (Gamma_2 * (F+1))/(T - O*(F+1)) - 1;