Semi-Asynchronous Checkpointing for Optimistic Parallel Simulation

Andrea Santoro
Semi-Asynchronous Checkpointing for Optimistic Parallel Simulation

Thesis Committee
Prof. Bruno Ciciani (Advisor)
Dott. Francesco Quaglia
Prof. Salvatore Tucci

Reviewers
Prof. Alois Ferscha
Prof. Philip Wilsey
AUTHOR’S ADDRESS:
Andrea Santoro
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma “La Sapienza”
Via Salaria 113, I-00198 Roma, Italy
E-MAIL: santoro@dis.uniroma1.it
WWW: http://www.dis.uniroma1.it/~santoroa/
“Though this be madness, yet there is method in’t”
Shakespeare, Hamlet (II,2)
Acknowledgements

Many people helped me to complete this dissertation. First of all I want to thank my advisor, Bruno Ciciani, for his support and useful suggestions during the whole Ph.D. period.

Furthermore I am certainly deeply indebted with Francesco Quaglia, with whom many results in this thesis have been obtained, for his patient tutoring and for serving as a member of my thesis committee.

I am also grateful to Prof. Alois Ferscha and Prof. Philip Wilsey for having accepted to serve as external referees of this dissertation, to Prof. Salvatore Tucci for having served as third member of my thesis committee, and to the coordinator of the Ph.D. program, Prof. Giorgio Ausiello.

I cannot forget the colleagues with whom I shared room 231 for so much time: Luigi Laura, Massimo Mecella, Giovanna Melideo, Marcelo Oglietti, Marco Emilio Poleggi, Enver Sangineti, Monica Scannapieco, Sara Tucci Piegovanni, Antonino Virgillito, Andrea Vitaletti and Fabio Zito. Their constant presence, jokes and laughs always prevented me from working too much. For the same reason I also thank Daniela Berardi, Andrea Cafà, Camil Demetriusc, Francesco Farinelli, Alberto Finzi, Domenico Lembo, Carlo Marchetti, Marco Pirrone, Massimo Romano.

A special thanks goes to Robert Heinlein, whose writings first showed me that there are more things in heaven and earth than are dreamt of in our philosophies. More in general, my thanks goes to all ancient and modern bards, from Homer to Tolkien passing through Shakespeare, for letting me have a glimpse of those philosophies (which are pretty vast on their own).

I certainly wish to thank my brother, Sergio, for his invaluable help and my parents, Italico and Silvana, for their support during good and bad times, for teaching me so much on so many subjects and for encouraging me in pursuing the subjects in which they could not coach me directly.

Finally a note of thank for what I learnt goes to all people I met in the course of years, with one unnamed exception (who knows perfectly well the reason).

And to everybody whom I am not likely to meet again, I say fare you well, friends...

.....until next time.

Roma, Italy

December 15th, 2002

Andrea Santoro
## Contents

Abstract 1

1 Introduction 5

1.1 Discrete Event Simulation ......................... 5
1.2 Parallel Discrete Event Simulation ................. 6
1.3 Maintaining Causality .............................. 6
1.3.1 Conservative Synchronization .................. 7
1.3.2 Optimistic Synchronization ..................... 8
1.3.3 Reduced Optimism Synchronization .......... 10

2 State Recovery in Optimistic Synchronization 13

2.1 Checkpointing ................................. 13
2.1.1 Sparse Methods ............................... 14
2.1.2 Incremental Methods ......................... 20
2.1.3 Hybrid Methods ............................... 23
2.1.4 Notes on Memory Concerns .................... 24
2.2 Reverse Computation ............................. 24
2.3 Observations ................................ 25

3 Semi-Asynchronous Checkpointing 27

3.1 System Model .................................. 28
3.2 Effectiveness Requirements ....................... 29
3.3 Data Consistency and Device Contention Issues .... 30
3.4 Re-Synchronization ............................. 30
3.4.1 Application Freezing Semantic (AF) ............ 31
3.4.2 Checkpoint Abort Semantic (CA) ............... 32
3.4.3 Conditional Checkpoint Abort Semantic (CCA) .... 33
3.4.4 Minimum Cost Semantic (MC) .................. 34
3.4.5 Comparative Discussion Between CCA and MC ...... 40
3.5 Remarks ................................ 41
4 An Implementation for Myrinet Clusters 43
  4.1 Myrinet Overview ........................................ 44
     4.1.1 Hardware Platform .................................. 44
     4.1.2 Typical Message Passing Layer .................. 46
  4.2 An Integrated Library for Checkpointing and Communication 48
     4.2.1 Software Architecture .............................. 50
     4.2.2 Communication Functionalities ................. 52
     4.2.3 Checkpointing Functionalities .................. 53
     4.2.4 Structure of the Extended Control Program .... 56
     4.2.5 Implementation of the \( \mathcal{C}A \) Re-synchronization Semantic 57
     4.2.6 Implementation of the \( \mathcal{MC} \) Re-synchronization Semantic 60
  4.3 Low Level Parameter Tuning ............................ 68
     4.3.1 Performance Issues Identification .............. 69
     4.3.2 Experimental Analysis ............................ 70
     4.3.3 Deductions ......................................... 74

5 Experimental Results 75
  5.1 Benchmark Study ....................................... 76
     5.1.1 Testing Methodology ............................... 76
     5.1.2 Part A ............................................. 78
     5.1.3 Part B ............................................. 85
  5.2 Study on a Real World Model .......................... 87

6 Conclusions ............................................... 99

Bibliography .................................................. 101
Abstract

Parallel discrete event simulation is a well known technique for design, tuning and evaluation of complex systems. It is used for simulation models which are too expensive in terms of computational power to be executed within reasonable computation time in a classical sequential fashion. In parallel discrete event simulation, the simulation model is partitioned into a set of separate Logical Processes (LPs) to be executed concurrently, each one modeling the evolution of a portion of the simulated system. In order to ensure correct simulation results, synchronization mechanisms are used to maintain a non-decreasing timestamp order for the execution of simulation events at each LP.

Optimistic synchronization defines a class of synchronization protocols capable of great exploitation of the parallelism intrinsic to simulation models. With these protocols, each LP is allowed to execute events as soon as they are available, with no verification on whether the execution will ultimately result in a timestamp order violation. In case a violation is detected, a rollback procedure is executed for recovering the LP state to a correct value. The state recovery procedure is typically based on logs of state information, known as checkpoints.

Since checkpoints are taken during the simulation execution, such a logging mechanism imposes an overhead which may even become unacceptable in case of simulation models with LPs characterized by very large states. (The largeness of the LP state is typically affected by the complexity of the system to be simulated, and also by the level of detail of the adopted simulation model for a given system to simulate.) Therefore checkpointing is a core problem to tackle in order to guarantee optimistic synchronization protocols to be really performance effective.

This dissertation presents an innovative mode to perform logging of state information in support of optimistic synchronization, namely semi-asynchronous checkpointing, based on the idea of real concurrency in the execution of checkpointing and other simulation specific operations. As compared to the classical synchronous mode, such an innovative mode poses a set of problems both from the conceptual and implementative standpoints. Solutions to these problems are also suggested and an implementation of a C library capable
to support semi-asynchronous checkpointing on off-the-shelf hardware widely used in the parallel computing community, namely myrinet based clusters, is presented.

An experimental study of semi-asynchronous checkpointing is also carried out, based on both classical parameterized synthetic benchmarks and real world application, namely a mobile communication system application. The results show that, except for the case of minimal state granularity applications, semi-asynchronous checkpointing definitely allows improvements of the speed of the parallel execution, as compared to classical synchronous checkpointing.

Chapter Organization

Chapters 1 and 2 are devoted to the introduction of parallel discrete event simulation, with a focus on optimistic synchronization and on existing results in the checkpointing area.

Innovative contributions of this dissertation start from Chapter 3, where semi-asynchronous checkpointing is presented. This is done by first introducing the model of a system capable to support such a checkpointing mode, with identification of requirements for its effectiveness, and then discussing issues related to such a mode. To tackle these issues, the concept of re-synchronization between checkpointing and other simulation specific operations is presented. Different re-synchronization semantics are also introduced, each one exhibiting a different simplicity/effectiveness tradeoff.

In Chapter 4 the implementation of the C library supporting semi-asynchronous checkpointing on a cluster interconnected by myrinet networks is presented. Finally, in Chapter 5, experimental results are reported.

Most of the material presented in this dissertation can also be found in the following technical articles:


A. Santoro and F. Quaglia, “Benefits from Semi-Asynchronous Check-


Chapter 1

Introduction

"The computer is actually a modelling device"
Douglas Adams, The salmon of doubt

In this chapter we describe Parallel Discrete Event Simulation (PDES) and the related terminology. Moreover, in Section 1.3 we point out the core issue in PDES, namely synchronization. Traditionally this problem has been solved by two different approaches, i.e. conservative and optimistic, which are in their turn recalled in Section 1.3.1 and Section 1.3.2, respectively.

1.1 Discrete Event Simulation

Discrete Event Simulation (DES) is one of the main techniques used for system design verification, system tuning and performance evaluation. A DES consists of a sequence of state changes or events, which replicates the relevant state transitions of the system to be simulated, known as the physical system. In other words, the sequence of state changes models the behavior of the physical system, thus being also referred to as simulation model for that system. The notion of time in the physical system is translated into simulation time, or logical time. Each event occurs at a discrete point in logical time and is therefore associated with a timestamp. The events must be executed in non-decreasing timestamp order to maintain the causal relationships between events in the simulated physical system.

A typical sequential DES simulator has three main components: (i) the event list, that is a list of events scheduled to be executed, ordered in increasing timestamp order; (ii) a logical clock, that is a value representing the time up to which the simulation model has been executed (typically it is the timestamp of the currently executed event, if any, otherwise it is the timestamp of the last executed event). Often the content of the logical clock is referred to as
Virtual Time; (iii) the state buffer, i.e. a set of variables describing the state of the simulated system at given points in simulation time. The simulator executes each event in the event list sequentially.

The execution of an event means: (a) advancing the logical clock at the time at which the event to be executed is scheduled, namely the timestamp of the event; (b) (possibly) modifying the state variables; (c) (possibly) scheduling new events to be executed at a later logical time.

Examples of physical systems widely studied by the means of DES are communication systems, queuing networks, logic circuits, and aviation control, among others.

1.2 Parallel Discrete Event Simulation

Despite of the potential of DES, many simulation models are so demanding in terms of computational resources that their execution might even be intractable if carried out on a single processor. A technique widely used to make complex simulation models tractable in practice consists in having many computers working together on the same model. This technique is typically referred to as Parallel Discrete Event Simulation (PDES) [33]. In PDES, the simulation model is partitioned into a set of entities, typically referred to as Logical Processes (LPs), each of which simulates the behavior of a specific portion of the physical system. Each LP is in essence a sequential DES simulator, with its own event list, logical clock, and state. The interaction between the different portions of the physical system to be simulated is represented by the exchange of timestamped messages between LPs along logical channels (no shared state is involved). In this way each LP may be run on a different processor, or even on a different machine. Typically, the receipt of a message at an LP has the effect to schedule one or more events at that LP.

Correctness of simulation results in the presence of such an interaction is ensured through adequate synchronization mechanisms among the LPs, as we shall discuss in the following section.

1.3 Maintaining Causality

A sufficient condition for the correctness of simulation results in a PDES is the execution of events in non-decreasing timestamp order at each LP [17], which is typically referred to as local causality, or simply causality. However, if no synchronization at all is used among the LPs, then this condition is not necessarily ensured. Consider the situation depicted in Figure 1.1, in which \( LP_1 \) and \( LP_2 \) proceed in simulation time independently of each other. \( LP_1 \) executes an event with timestamp 100, while \( LP_2 \) is executing an event with
timestamp 130. At the end of the event execution $LP_i$ sends a message to $LP_j$ to schedule an event $e$ with a timestamp of 110. Thus $LP_j$ will eventually receive a scheduled event with timestamp lower than the timestamp of already executed events. Processing $e$ after its receipt means $LP_j$ is not executing events in the correct order, therefore local causality is violated. In order to solve this problem, two main approaches to synchronization have been envisaged, i.e. conservative and optimistic. We shall discuss these approaches in the following sections.

### 1.3.1 Conservative Synchronization

A kind of protocols to avoid causality violations are conservative. In such protocols each LP executes the next event in its event list only when it is sure that the event execution is safe, i.e. event execution will not eventually result in any timestamp order violation. The basic assumptions underlying this kind of protocols are that communication channels between LPs preserve the order of messages, i.e. messages are delivered to the recipient LP in the same order they have been sent, and that messages sent along the channel between $LP_i$ and $LP_j$, if any, carry events with non-decreasing timestamps. Under these assumptions, in case there is at least a message in each input channel at an LP, then the message with the minimum timestamp carries a safe event. Therefore the LP can extract that message from the corresponding channel and execute the associated event.

The main problem that arises with this approach is deadlock since a cy-
cycle of LPs may block event execution waiting for messages from each other. Specifically, when LPs in a cycle of the communication graph all have at least one input channel with no incoming message, none of these LP is allowed to execute simulation events. Thus no LP will eventually send a message to another LP in order to allow the recipient to remove the block condition and to proceed in the execution of events.

Most conservative protocols allow deadlock avoidance, or deadlock recovery, through the exchange of proper protocol messages, which contain enough information to ascertain event safety even in the absence of application messages carrying simulation events along a channel between LPs. Examples of conservative synchronization protocols are the Null-Message Protocol [17], Deadlock Detection and Recovery [18], SRADS [79] and Appointments [63]. The notion of lookahead plays a fundamental role in the reduction of the amount of protocol messages to be exchanged. (Informally, an LP with simulation clock $T$ has lookahead $L$ if it can predict lack of output events up to simulation time $T + L$ on the basis of incoming events up to time $T$.) Specifically, the larger the lookahead, the more effectively an LP can detect event safety since each protocol message may carry information about absence of events along a given channel for a wide interval of simulation time.

These protocols have revealed the potential for high effectiveness in specific application areas, such as the simulation of road traffic [57], of switching networks [19] and of communication networks [70]. However, by their nature, they suffer from missing of opportunities to exploit parallelism (this is because an event might be actually safe even if safety has not been yet detected by the LP) and also from restrictions related to (i) predefined of the communication graph among the LPs and (ii) non-decreasing timestamps for messages exchanged along each given channel. Also, the overhead due to the protocol messages depends on the amount of lookahead present in the simulation model. As a consequence, these protocols may exhibit a limited speedup on particular simulation applications, or might even be useless when previous restrictions cannot be met.

### 1.3.2 Optimistic Synchronization

As opposed to conservative solutions, in optimistic synchronization protocols an LP is allowed to execute simulation events whenever they are available, by optimistically assuming that the execution will not eventually violate timestamp order. Should a real violation actually occur, a rollback procedure is executed in order to recover the LP state to a previous correct value (by “correct state value” we mean a state value generated by a computation that meets local causality constraints). An example of execution pattern under the Time Warp protocol [40, 41], the most known optimistic synchronization protocol,
is schematized in Figure 1.2. Each LP executes events in its event list without checking for their safety and advances its logical clock accordingly (the value stored into the logical clock is typically known as Local Virtual Time or LVT). New events are scheduled for other LPs by sending what are called positive messages. However, if a message from \( LP_j \) arrives in the logical past of \( LP_i \), i.e. its timestamp is lower than the LVT of \( LP_i \), the message arrival reveals a violation of local causality (usually such a message is referred to as straggler). To recover from the violation, \( LP_i \) performs a rollback procedure, which means: (i) recovering the state of \( LP_i \) at the value it assumed just before the logical time of the straggler message; this implicitly means that some events are rolled back since some state transitions are actually discarded while performing state recovery. (ii) sending out antimesages, i.e. negative messages, to annihilate (i.e. cancel) the positive messages sent during the rolled back computation, namely those messages sent as the effect of the execution of rolled back events. When an antimessage arrives, which cancels an event already executed, the recipient LP rolls back as well. This phenomenon is known as cascading rollback.

\[ 
\text{Events} 
\begin{array}{c}
\text{LP}_i \downarrow 90 \quad 110 \quad 120 \\
\text{LP}_k \downarrow 115+ \quad 115- \\
\text{positive message} \\
\text{LP}_k \\
\end{array}
\]

Figure 1.2: An Example of Execution Pattern Under the Time Warp Protocol.

A relevant concept to optimistic synchronization protocols is Global Virtual Time (GVT), which is defined as the smallest timestamp among the following: (i) unexecuted events already inserted into the LP event lists; (ii) events being executed; (iii) messages/antimessages in transit. Since no LP can ever rollback to simulation time preceding GVT, the GVT value indicates the commitment horizon of the simulation. It is used both to execute actions that cannot be subject to rollback, such as displaying of intermediate simulation results, and also for recovering memory. Specifically, events with timestamp lower than GVT will not eventually need to be re-executed after a rollback, therefore they
can be discarded. Also, obsolete information, if any, maintained in order to support state recovery can be discarded since state recovery will not eventually involve state values related to simulation time preceding GVT. Recovering memory after GVT calculation is typically referred to as *fossil collection*.

Several well known implementations of the Time Warp protocol exist. Among others we signal the Jade Time Warp environment [3], the SPEEDES (Synchronous Parallel Environment for Emulation and Discrete Event Simulation) operating system [90] (which also supports some of the reduced optimism synchronization protocols described below, in Section 1.3.3), the WARPED simulation kernel [54] and the GeorgiaTech Time Warp (GTW) [23].

Since LPs may execute simulation events as soon as they are received instead of blocking until event safety detection, optimistic protocols have greater potential for exploitation of parallelism as compared to conservative ones. Such a potential really yields large speedups when the total costs of both rollbacks and the associated operations to support it (e.g. maintenance if information to support state recovery during rollback) are kept low. Success of these protocols has been observed for simulations of communication networks [13, 14, 15, 51], queuing networks [71], Petri nets [28], aviation control [97], and digital logic circuits [9], among many others.

### 1.3.3 Reduced Optimism Synchronization

An optimistic approach is believed to be more conducive to an efficient synchronization protocol [33], since conservative protocols set boundaries on the amount of exploitable parallelism. However, as already mentioned, adequate speedup can be achieved in case the cost of rollback is kept low and the cost of any operation needed to support rollback is also kept low. Reduction of these costs is therefore the primary objective of research in the field of optimistic synchronization. Along one research direction we find the so called reduced, or controlled, optimism synchronization protocols, which attempt to reduce the rollback cost by reducing the amount of rollback in the parallel execution. This is achieved just by reducing the level of optimism in the execution of simulation events, i.e. imposing a sort of control on optimism. These protocols have been classified in [86] as follows:

- **Window based** [26, 50, 55, 78, 84, 91, 93]: each LP optimistically executes events only if their timestamps are inside an agreed-upon window.

- **Space based** [1, 7, 25, 37, 56, 77, 89]: LPs are partitioned into clusters. Interactions, i.e. event scheduling, inside the same cluster are treated optimistically, while interactions between different clusters trigger a conservative behavior.
- **Penalty based** [4, 78]: LPs are penalized (i.e. temporarily blocked) or favored (i.e. optimistically execute events) depending on their recent rollback behavior.

- **Knowledge based** [53, 72]: as soon as a computation is determined to be incorrect, the affected LPs limit their optimistic processing.

- **Probabilistic** [52]: a special process periodically sends out a probabilistically determined message to all LPs, forcing them to re-synchronize, i.e. to rollback if required, at the message timestamp.

- **State based** [22, 27, 30, 39, 49, 87]: LPs take adaptive decisions about their own optimism (i.e. about whether to temporarily block or optimistically proceed) by gathering, directly or indirectly, state information about other LPs.

These protocols, being optimistic at least to a limited extent, still have to cope with reduction of the cost of each single rollback operation and also reduction of the cost of operations needed to support rollback. The reduction of the costs to maintain information for state recovery, and to perform state recovery itself during a rollback phase, is the focus of this dissertation. Such a reduction has been explored in research directions orthogonal to reduced optimism. We survey the relative results in Chapter 2.
Chapter 2

State Recovery in Optimistic Synchronization

“If I have seen further than others it is only because I have been standing on the shoulders of giants.”
Sir Isaac Newton

As recalled in the previous chapter, parallel simulators based on optimistic synchronization protocols employ rollback mechanisms to recover from causality violations by reconstructing a previous LP state. These mechanisms impose an overhead on the parallel execution due to the following reasons: (i) Reconstruction of a state is not cost free. There is a state recovery cost to be paid anytime the simulation actually rolls back; (ii) Depending on the specific state reconstruction technique, some information about previous states and/or event executions might be needed to rollback. Also saving such information might be a costly operation.

Two main techniques are employed to support state recovery operations. The first one consists in saving (in whole or in part) the LP state during the computation progress. This technique is usually referred to as checkpointing. Upon a rollback, the state is recovered by exploiting the saved portions of the state, known as checkpoints. The second one consists in performing the inverses of the operations that were executed during the event computation, namely reverse computation. This chapter presents a survey on the existing literature about techniques to support state recovery.

2.1 Checkpointing

The traditional approach to implement rollback recovery involves the use of logs of state information, namely checkpoints, maintained at the simulation
application level. The action of taking the log of an LP state is usually referred to as checkpoin
ting operation.

The simplest way to use checkpointing in support of state recovery is to take a checkpoint of the whole state of an LP prior to the execution of each event [40]. This technique is typically known as copy state saving. Using this technique, state recovery of a previously traversed state consists simply in copying back the corresponding checkpoint into the LP state buffer. Needless to say, checkpointing the LP state each time a simulation event is executed might exhibit a relevant overhead. This is true especially in case of simulations with large states such as battlefield and large communication systems simulations (see for example [81]).

Several methods have been proposed to reduce checkpointing overhead as compared to copy state saving. Typically they are categorized in two different classes, namely sparse and incremental.

2.1.1 Sparse Methods

An attempt to tackle the checkpointing overhead problem is the adoption of what are typically referred to as infrequent, or sparse, checkpointing methods [6]. To avoid the overhead of taking a checkpoint at each event execution, only a subset of the states traversed by the LP during the execution are checkpointed. In case the simulation must rollback to a state that was not checkpointed (missing state), the state to be recovered must be reconstructed. This is accomplished by reloading a previously checkpointed state and then replaying the computation of intermediate events in order to update state variables. This procedure, known as coasting forward, originates an increase in the state recovery overhead as compared to state recovery when copy state saving is adopted. This is due to the additional computation for replaying coasting forward events.

It is important to remark that such an approach is feasible in case of simulation software whose event computation can be deterministically replayed. Otherwise, the reconstructed state and the one we are attempting to reconstruct might diverge after the execution of coasting forward events. Anyway, it is well known that employing pseudo-random generators of adequate size to emulate non-deterministic computation, and saving their seeds as a part of the LP state, allows any coasting forward event to produce the same updates on the LP state variables as the original execution during normal, forward computation. The usage of such a kind of pseudo-random generators is widely considered as a standard in the DES technology (see for example simulation tools like OPNET [65], OMNeT++ [96], NS [64], GTW [23], WARPED [54] and SPEEDES [90]), thus making sparse checkpointing methods practical solutions to reduce the overhead of copy state saving.
2.1. CHECKPOINTING

The main problem that has been studied for sparse checkpointing is the selection of the states to be checkpointed. Based on considerations related to checkpointing techniques in support of fault tolerance, the approach in [6] suggested to use the amount of CPU time used by an LP to determine a timeout for taking a checkpoint of the LP state. Solutions specifically tailored for the PDES field suggested to checkpoint the LP state periodically each \( \chi \) event executions, where \( \chi \) is typically known as the checkpoint interval of the LP. In this case, it is of paramount importance to select an appropriate value for \( \chi \). Checkpointing too frequently (i.e. \( \chi \) too small) might lead to significant checkpointing cost, especially when the LP state is relatively large. On the other hand, checkpointing too infrequently (i.e. \( \chi \) too large) is likely to give rise to an excessive increase of the length of coasting forward. Such an increase negatively affects performance especially for frequent rollbacks in the parallel execution, and when the granularity of coasting forward events is non-minimal. Overall, \( \chi \) should always be set to the value that realizes the best trade-off between checkpointing and recovery overheads as a function of both real execution costs (e.g. event granularity and state saving cost) and the rollback pattern of the specific execution. As an extreme, the best \( \chi \) for an LP that never rolls back would be infinity, since, it eliminates the checkpointing overhead while still allowing no coasting forward overhead to be paid due to absence of state recovery procedures to be executed \(^1\). Also, since the rollback behavior might change dynamically, the well suited value for \( \chi \) might even vary in the course of any single parallel execution, thus making more difficult its identification.

Several works, here discussed, have tackled the problem of the checkpoint interval selection. Some of them attempted to estimate the value of \( \chi \) by modeling the combined cost of checkpointing and state recovery as a function of \( \chi \). Classical assumptions underlying these models are: (i) The latency of checkpointing a state, or equivalently, copying it back, exhibits almost no variance. This assumption is typically considered realistic since such a latency consists merely in transferring a certain amount of bytes between memory buffers. (ii) The latency to replay an event during coasting forward can be reasonably approximated by the expected event granularity. This assumption is reasonable anytime the variance of event granularity is relatively low. (iii) There is no correlation between where rollbacks occur in simulation time and the positions of checkpoints. This assumption has been validated by experimental results in [31, 71, 80] which confirm the intuitive consideration that, in case of periodic checkpoints taken each \( \chi \) events, the coasting forward length, in terms of number of coasting forward events, is uniformly distributed in the interval

\(^1\)There is an objection to the use of an infinite \( \chi \) which is related to the overhead due to memory management. It will be discussed in Section 2.1.4.
[0, \chi - 1], thus exhibiting an expected length of \( \frac{1}{2^{\chi - 1}} \) events.

An example of this kind of models can be found in [48], where the combined checkpointing and recovery overhead is modeled in a way to allow the identification of a lower bound (\( \chi^- \)) and an upper bound (\( \chi^+ \)) for \( \chi \) \(^2\). This model relies on the assumption that the LP rollback pattern (e.g. the frequency of rollbacks experienced by that LP) is not affected by changes in the checkpoint interval. As respect to this assumption, the extended experimental analysis in [71] shows that it may not be verified in some circumstances. Specifically, a variation of \( \chi \) typically leads to a variation in the state recovery latency which in its turn might have two opposite effects on the amount of rollback:

- For a small increase of \( \chi \) the amount of rollback may decrease. This happens because the slight increase in the state recovery latency, due to slightly longer coasting forward, holds the LPs from executing for a while when state recovery is carried out on a given processor. Therefore more messages/antimessages for those LPs may arrive carrying/cancelling scheduled simulation events. This allows the simulation engine to take "better" processor scheduling decisions while selecting the LP the processor must be assigned to. This effect is known as rollback throttling.

- For a very large increase of \( \chi \) the amount of rollback may increase. Specifically, for very large values of the checkpoint interval the state recovery latency might be relevantly increased due to non-minimal length of coasting forward, thus the differences between the LVTs of LPs on different machines might increase when state recovery is carried out on a given machine. More precisely, LPs on a machine may remain blocked for a non-minimal interval, due to the execution of the state recovery procedure on that machine, while LPs on another machine might proceed in simulation time. Therefore the likelihood that a message/antimessage from the former LPs may trigger a rollback on LPs on the other machine is likewise increased. This may lead to a significant increase in the number of rollbacks, which is referred to as rollback thrashing.

To take into account the potential variability of the rollback behavior vs \( \chi \), in [48] \( \chi^- \) and \( \chi^+ \) are iteratively recomputed and a tentative \( \chi \) between the two bounds is selected at each iteration, until \( \chi \) itself converges. However there are also other reasons for which the rollback behavior may vary during the simulation itself. For instance the load on one or more machines may vary over time. Or the average granularity of events occurring at some LPs

\(^2\)A single value for \( \chi \) could not be determined, since the model in [48] could be solved only in case the probability distributions of both the number of executed events between two rollbacks and the rollback length (i.e. the number of events undone on an LP by a single rollback) were known.
on one machine may change with the simulation progress. In both cases the
LPs hosted on some machines tend to stay behind the others in the virtual
time, therefore their messages/antimessages are more likely to trigger roll-
backs. Moreover, depending on the simulation model, the frequency of mes-
sage exchange among the LPs may vary over time, thus possibly producing
variations of the frequency of rollback. To cope with any source of variation
for the rollback behavior, the algorithm above should be periodically run to
maintain an optimal $\chi$ value in the lifetime of the simulation.

The works in [67, 68] have used a slightly different model, incorporating
some additional details about the rollback behavior, to derive a single opti-
mal value for $\chi$ ($\chi_{opt}$), instead of two separate boundaries. Specifically, [48]
derives $\chi^-$ and $\chi^+$ using the average number of events executed between con-
secutive rollbacks, while [67] derives $\chi_{opt}$ as a function of the average number
of rollbacks and the average rollback length. However since the number of
committed events is also necessary to derive $\chi_{opt}$, the frequency of updating
the checkpoint interval cannot be higher than the frequency of GVT calcula-
tions (by definition an event is committed only when its timestamp is lower
than GVT, see Section 1.3.2).

A solution to adaptively calculate an optimal value for $\chi$ is presented in
[80]. This work introduces a model from which an expression for $\chi$ is derived
by including only quantities that can be easily monitored during the execution,
that is the total number of executed events (committed plus rolled back) and
the number of rollbacks. By using this model, calculations of $\chi$ can take
place at any time, thus efficiently supporting adaptive re-calculation of the
checkpoint interval $\chi$ for any individual LP. In addition [80] also proposes
a method to decide when re-calculation should take place. This is done by
deriving the amount of observations (on the monitored quantities) required
for a 90% confidence on a small interval around the optimal value of $\chi$. [80]
concludes that a re-calculation each few hundred events is typically sufficient
for a rapid convergence of $\chi$ to the real optimal value. This checkpointing
method has been integrated into MobSim++, a tool for large-scale simulations
of personal communication networks [46].

A different approach to the problem of selecting $\chi$, not relying on analytical
models, is presented in [31]. This work presents a simple (and therefore fast
to compute) heuristic to adaptively re-calculate the value of $\chi$. The heuristic
is based on the consideration that the checkpointing overhead ($C_{SS}$) decreases
monotonically with $\chi$, while, as already observed in [71], the overhead of coast-
ing forward ($C_{CF}$), and thus the overhead of state recovery, increases linearly
with $\chi$. Therefore, the combined overhead of checkpointing and coasting for-
ward ($C_{SS} + C_{CF}$) assumes the characteristic shape displayed in Figure 2.1,
with a clear absolute minimum and no other local minima. Exploiting such
a shape, the authors derive the following heuristic for adjusting $\chi$ during the
Figure 2.1: Combined Overhead of Checkpointing and Coasting Forward.

execution:

1. $\chi$ is initially set at one (i.e. the LP initially behaves as if copy state saving were adopted);

2. Time spent performing checkpointing and coasting forward during a fixed number $N$ of executed events is monitored and stored;

3. $\chi$ is incremented by one;

4. Time spent performing checkpointing and coasting forward during a fixed number $N$ of executed events is again monitored, stored and confronted with that stored at the end of the last monitoring phase. In case the last monitored time is less than the previously monitored one, $\chi$ is incremented by one, otherwise $\chi$ is decremented by one.

The last step is iterated until the simulation completion. As an additional safeguard to prevent oscillations in the checkpoint interval an appropriate threshold in the difference between the two compared values is adopted to determine whether $\chi$ must really be adjusted. As respect to the determination of $N$, [31] suggests to estimate it as proposed in [80]. This heuristic has been implemented in the WARPED simulation kernel [54], and has been proven to provide relevant performance improvements for simulations of digital systems described according to the VHDL language [69].

A relevant observation, exploited in [83], is that previously developed models have taken into consideration only the expected event granularity while modeling coasting forward overhead associated with state recovery. Whenever the event granularity variance is large, those models might yield a sub-optimal
determination of the checkpoint interval $\chi$. Therefore [83] classifies different types of events in classes, according to their own coasting forward times and, according to a model, assigns to each event class a different probability for taking a checkpoint after the event execution. More specifically events with longer coasting forward times should be checkpointed more frequently due to the longer overhead they impose in case they must be coasted forward (i.e. re-computed). Although only a manual tuning of such class-related checkpointing probabilities is presented, a mechanism for the class-related checkpointing probabilities selection should not be too difficult to design. The main drawback of this method is that different event classes must be explicitly identified by the simulation designer. Also, a large variance in the execution time of a single event type cannot be captured by this mechanism. The technique has been tested on the simulation of a mobile telecommunication network modeling a ring highway with uniformly distributed base stations. Promising results have been obtained on that simulation model.

Another model for the checkpointing and state recovery overhead vs $\chi$ has been presented in [2], which is well suited when memory space for the entries of the stack of checkpoints are allocated/deallocated dynamically (e.g. by using the `malloc()` - `free()` mechanism of C language). This model derives an expression for the optimal value of $\chi$ by considering also the overhead due to low level parameters related to the checkpoint stack management performed during state recovery procedures.

More recent works have also started to challenge the assumption that checkpoints should be taken periodically each $\chi$ event executions, independently of the way in which $\chi$ itself is determined, and explored non-periodic approaches to the selection of states to be checkpointed. For example, on the basis of the assumption that the likelihood of a rollback point being located between two consecutive events increases with the virtual time distance between them, the work in [73] presents a checkpointing rule that forces the LP to record its state prior to an LVT increment which oversteps a given threshold. The threshold is defined as $\alpha \Delta_{LVT}$ where $\Delta_{LVT}$ is the average distance between the timestamp of consecutive events and $\alpha$ is a value greater than or equal to zero, selected dynamically through a heuristic similar to that underlying the technique for the adaptive re-calculation of $\chi$ presented in [31]. Although the basic assumption underlying this solution (i.e. the likelihood of a rollback occurring between two consecutive events increases with the virtual time distance between them) has been never extensively tested, performance improvements have been noted for the case of a standard set of synthetic simulation benchmarks.

[74] presents an heuristic, namely probabilistic checkpointing, relying on an estimation method for the probability of a rollback occurring between two consecutive events. This method is based on splitting the domain of the pos-
sible $\Delta_{LVT}$ (as defined above) in separate ranges and observing the rollback rate for each range of $\Delta_{LVT}$. According to the estimated rollback probability, a checkpoint is taken with the same probability. In other words, the higher the probability for a rollback to occur between two events, the higher the likelihood that the LP state is checkpointed before executing the second event, which is expected to reduce the coasting forward length. However in case very long sequences of executed events exhibited very low rollback probabilities, we might be taking very few checkpoints. Therefore extreme situations in which the LP executes very long sequences of uncheckpointed events may arise, thus possibly leading to loss of control on the state recovery cost in case of rollback to an event in that sequence. To avoid this problem, probabilistic checkpointing is merged with a classical periodic checkpointing technique where $\chi$ is estimated as suggested by [31]. This technique has been tested with positive performance results on simulations of a communication network with torus topology and store-and-forward packet switching.

Finally, in [76] a checkpointing/recovery cost model has been presented which, given the current state, determines the convenience of recording it as a checkpoint before the next event is executed. This is done by taking into account the position of the last taken checkpoint, the cost of taking the checkpoint, the real granularity (i.e. the execution time) of intermediate events from the last taken checkpoint, and using an estimate of the probability that the current state will have to be eventually restored due to rollback in the future of the execution. In other words, this model allows to take a fine decision, based on (predicted) costs, on whether each single state should be checkpointed. Beyond reporting performance results for the case of a mobile system simulation application, this work also presents a discussion on when periodic and non-periodic sparse checkpointing techniques are expected to be effective in practice. The discussion points out that periodic solutions have the potential for being as effective as non-periodic solutions except for the case of a given type of rollback patterns in the parallel execution.

### 2.1.2 Incremental Methods

Another approach to the reduction of the checkpointing overhead is given by incremental methods [5]. Such methods do not checkpoint the whole state, but only the portion of the state that is actually going to be modified by the next event, thus reducing (on a degree dependent on the specific simulation model) the amount of data saved at each event execution, i.e. the checkpointing cost per event. In this case, state recovery to a previous state is accomplished by backward traversing the incremental checkpoint history and copying back into their original locations each of the previously saved state portions.

Practical implementations of incremental checkpointing can be found in
the literature. Specifically, [94] presents an implementation of incremental checkpointing for the Jade Time Warp environment [3]. This paper presents what is termed as an External Management System (EMS) introduced to separate the simulation model from the specific checkpointing functions, that might differ for different kinds of data and for different patterns of access to those data. In addition, two different implementations of incremental checkpointing [92] to be used in object oriented software have been embedded into the SPEEDES operating system [90]. The first one, termed Delta Exchange, consists in maintaining a list of executed events as separate objects, each one with its own state. When an event is executed, the updated state variables are exchanged with corresponding variables into a unique simulation object. The second (more complex) implementation, termed Rollback Queue, copes with situations where the simpler Delta Exchange cannot work (an example is when an event modifies a dynamically allocated memory structure), and consists in maintaining a queue listing all the changes occurred to data structures at each event execution. Rollback Queue should be used only when necessary since it is much slower than Delta Exchange.

The main drawback of incremental checkpointing consists in the fact that its performance is mostly dependent on features of the simulation model and also on the specific rollback pattern. Specifically, incremental checkpointing achieves good performance in case of simulations whose state is updated at each event only by a small fraction and whose average rollback length (i.e. the number of events rolled back by a single rollback occurrence) is short. In these situations, the overhead of checkpointing is low and the state recovery cost is low as well since recovering the LP state needs to copy back few and small state portions. However, on the opposite side we may find simulations whose state is largely updated by each event and whose average rollback length is long. In this kind of simulations incremental checkpointing is likely to exhibit performance worse than sparse checkpointing or even copy state saving, since the overhead of checkpointing is almost the same as for copy state saving, while recovering a state consists in copying back a number of state portions equivalent to the number of events rolled back (instead of simply copying back a single checkpointed state, as it would be in copy state saving).

The absence of tunable parameters for controlling the combined overhead of checkpointing and state recovery, as for example the checkpoint interval \( \chi \) of periodic checkpointing, explains why modeling incremental checkpointing has not received as much attention as sparse checkpointing. However some models [20, 67] have been developed with the aim at analytically comparing incremental and periodic checkpointing methods, and therefore help the simulation designer to choose the checkpointing method best suited for the specific simulation application. According to these models, the situation in which incremental checkpointing outperforms periodic checkpointing is (as it could be
expected) when the fraction of the state saved at each event execution is small.

Other works attempted to tackle the transparency problem inherent to incremental methods. More specifically, identifying each change of the LP state in a parallel simulation is not as easy as it would be in transaction oriented applications (e.g. databases) where each state modification is explicitly filed to a separate entity. The simplest approach to this problem consists in creating functions capable to save the value of a single variable but that must be inserted by the programmer before the modification of the state variable. This approach increases the source code complexity and therefore it is error-prone and difficult to debug.

An alternative approach includes the use of augmented data types to improve transparency. For example, [10] improves transparency by using what in the data structure communities are known as persistent objects, a kind of data structures that “remember their past”. In contrast to conventional data structures, whose state is destroyed by any change and only the new state can be further accessed, persistent objects allow access to any version of the data structure, past or present, thus making them ideal candidates to implement an LP state when incremental checkpointing must be used. Another way to improve transparency has been proposed by [81] where overloaded C++ operators are employed to the same goal. Specifically, since in C++ any state modification takes place by using an operator like -=, +=, +=, etc., the backup method can be invoked each time the state is modified by overloading such operators for the objects to be checkpointed. However, also this approach, as the previous one, cannot stand the adoption of third part libraries not specifically designed to support incremental checkpointing. This technique has been tested for a mobile phone system simulation application. The results in case of large state sizes show a speedup very near to manual insertion.

The most transparent approach has been proposed in [95]. With this solution the executable code can be automatically modified to support incremental checkpointing using the Wisconsin Architectural Research Tool Set (WARTS) Executable Editing Library (EEL) [45]. Although this technique ensures transparency, its application is limited to the architectures supported by the executable editing tool (the executable format changes with the operating system while the executable code itself depends on both the operating system and the hardware platform).

Finally, [35] presents the design and the implementation of a special purpose hardware, namely the Rollback-Chip, to be mapped into the host memory address space, capable to support incremental checkpointing. The rollback chip maintains so called version controlled memory entries, capable of logging versions of specific portions of the LP state. The action of logging a state portion within a given entry takes place upon the execution of a MARK command. As an extreme this chip is capable to checkpoint the whole LP state.
This can be accomplished invoking the MARK command for all the LP state portions at a specific point in simulation time. In other words, this chip can be seen as a combination of a memory management unit and a data cache specifically tailored to the need of checkpointing for optimistic simulation.

2.1.3 Hybrid Methods

Other checkpointing methods are also worth mentioning though they are not easily categorized as sparse or incremental, since they tend to merge characteristics of both classes.

In [38] a multiplexed checkpointing method is presented, in which checkpointing is usually performed according to an incremental method (i.e. only updated state variables are checkpointed), but in addition a checkpoint of the whole state is periodically taken. State recovery to virtual time $t$ is realized by reloading the earliest whole-state checkpoint with virtual time greater than $t$ into the LP state buffer, and then backwardly applying the state changes starting from the virtual time of the reloaded checkpoint. This method has been successfully applied to interactive simulations [32], where users are allowed to query for state values related to the far back of the execution. Reconstructing those values with classical incremental methods, i.e. starting from the current simulation time and traversing the incremental checkpoint history, would lead to unacceptable latency for providing results to the end user. A performance model for this checkpointing method has also been presented in [85].

Beyond interactivity, from the execution speed standpoint this method may work fine for simulations with long rollback length, but it just boils down to simple incremental checkpointing in case of simulations with short rollback length. This problem is tackled by a more general hybrid method [21], which performs both incremental and sparse checkpointing in a fashion similar to the multiplexed method, but only a subset of the events between two whole-state checkpoints are actually incrementally checkpointed. Specifically, whole-state checkpoints are taken according to the classical checkpoint interval $\chi$, while incremental checkpoints are not taken for any event whose state is not completely checkpointed but only for $\nu$ events immediately preceding a whole-state checkpoint. Therefore in each interval between checkpoints there will be some events whose state is completely uncheckpointed. In this context also the state recovery technique is hybrid. Actually, depending on whether the state to be restored has been incrementally saved, state recovery to virtual time $t$ may consist in reloading the latest whole-state checkpoint before time $t$ and coasting forward up to time $t$ or reloading the earlier whole-state checkpoint after time $t$ and backwardly applying state changes up to time $t$. 
2.1.4 Notes on Memory Concerns

Up to now we have considered only costs that various checkpointing techniques have on the completion time of the simulation, implicitly assuming that the system supporting the simulation model execution has unbounded memory to accommodate all checkpoints. However, checkpointing techniques do not only use CPU resources (i.e. time) but also memory (i.e. space). In other words checkpointing exhibits a cost not only in time but also in space.

From the point of view of memory consumption copy state saving is the most expensive checkpointing method, since the whole state is saved at each event execution. Whenever sparse checkpointing is employed, [71] shows that, as it is expected, memory consumption due to checkpointed states is highly reduced. However for very large values of the distance between checkpoints a different effect should be considered. Specifically, in order to correctly execute rollback operations, an LP must be able to recover any state with simulation time in between GVT and its current LVT. With sparse checkpointing this is accomplished maintaining the last checkpointed state with simulation time less than or equal to GVT and also all the events with timestamp larger than the simulation time of that checkpoint, since they might need to be re-executed in a coasting forward phase. In other words, these events cannot be fossil collected. Consequently, as the distance between checkpoints is largely increased, also the memory required to store events increases.

On the other hand, memory consumption of incremental checkpointing is strictly related to the specific simulation model. For a model in which the LP updates the most part of its state at each event execution, incremental checkpointing is likely to require almost as much memory as copy state saving, while the opposite is true for models in which the LP updates only a small fraction of its state at each event.

Typically memory is a low-cost commodity, therefore space costs are often less relevant as respect to time costs. Whenever this assumption is not verified care should be taken in selecting the adequate checkpointing technique, and also its parameters, in order not to significantly slow down the whole simulation due to reduced effectiveness of the underlying virtual memory system. In case of sparse checkpointing, discussions on how to select the maximum distance between checkpoints in order not to incur performance problems due to memory consumption can be found in [31, 71, 80].

2.2 Reverse Computation

A different approach to state recovery has been studied in the literature, namely reverse computation. Instead of recovering a previously traversed state by copying a checkpoint back into the state buffer (and possibly performing
coasting forward) or by backward retraversing incremental checkpoints, reverse computation consists in the simulation application performing the inverses of the operations that were executed during the event computation itself. Although application of this technique has been deemed as inviable in practice until recently, due to the practical difficulty of creating not only a reversible event execution code but also the code to actually reverse its computation, [16] has shown how compiler supports can automatically generate both the reversible and the reversing code.

Reverse computation apparently avoids the usage of state information logs, which is typical of checkpointing. However, depending on the structure of the event to be reversed some information on the event execution flow might be needed to correctly reverse the computation. As an example, some data are required to reverse if statements modifying their own predicate. Depending on the complexity of the event structure, this information might be smaller or larger than the state itself. Should the event be very complex, classical checkpointing would require to save less information than the data required to reverse the event computation. Moreover some operators are simply non-reversible (e.g. plain assignments), and the only way to reverse them is to checkpoint the variables on which they are executed.

All these drawbacks sum up in the consideration that reverse computation is very sensitive to the structure of simulation events. Although in case of simple events containing reversible code reverse computation is very likely to be conducive to performance improvements, quite the opposite can be said in case of complex events containing non-reversible code.

In addition, to restore a state we need to perform a reverse computation of all the events executed between the current one and the one we want to roll back to. The longer is the rollback length, the heavier will be the rollback cost (this dependency is similar to the increase in the rollback cost vs the rollback length in incremental checkpointing).

For all the reasons outlined above, the mainstream in the field of PDES is still oriented towards checkpointing as the preferred way to support state recovery (the authors of [16] themselves propose reverse computation only as a complement to classical checkpointing).

### 2.3 Observations

Almost any checkpointing overhead reduction method, whether it be sparse, incremental or hybrid checkpointing, attempts to reduce the cost of checkpointing by reducing the total amount of saved information per event execution as respect to copy state saving. In certain situations the presence of memory constraints may even be a good reason to reduce the amount of in-
formation saved per event execution. However, whenever memory is not a concern, such a reduction has an undesirable increase in the expected cost of state recovery. As an example, this happens due to coasting forward when sparse state saving is adopted.

Few attempts have been done in an orthogonal direction, that is to reduce cost of a checkpointing operation without reducing the amount of information checkpointed through that operation. Two results tackling the checkpointing problem from such a standpoint are the hardware acceleration obtained by the Rollback-Chip, already described in Section 2.1.2, and a technique known as Fast-Software-Checkpointing [75]. With this technique, part of the instructions performed during event execution are actually used as part of the checkpointing protocol in a way to reduce the total amount of instructions explicitly used to checkpoint the LP state. In other words, the checkpointing protocol is partially embedded within the event execution code. This has the potential to allow a reduction of the latency of any single checkpointing operation with no reduction of the amount of checkpointed information through that operation. The effectiveness of this approach is dependent on the degree at which the previous embedding can be realized, thus it is dependent on the structure of the event routine code. Another major issue relates to reduced transparency since, unless compiler supports are available, the responsibility to embed the checkpointing protocol within the event routine code pertains to the application programmer.

This dissertation is focused exactly on the reduction of the overhead of any single checkpointing operation while attempting not to reduce the amount of checkpointed information through that operation. This will be accomplished through the notion of semi-asynchronous checkpointing we shall present in Chapter 3. As we will show in Chapter 4, semi-asynchronous checkpointing can be actually implemented on standard, off-the-shelf hardware, thus revealing itself a solution different in nature as respect to the Rollback-Chip. Also, transparency at the application programming level is maintained, since no special care must be taken, as respect to the checkpointing mechanism, while implementing the event routine code.
Chapter 3

Semi-Asynchronous Checkpointing

"Time is an illusion."
Douglas Adams, The hitchhiker’s guide to the galaxy

So far, both incremental and non-incremental checkpointing methods are characterized by the fact that checkpoints are taken in accordance with the synchronous execution model. Specifically, any activity in the simulation application is suspended upon issuing the checkpointing request, and is resumed only after the checkpointing protocol completes (i.e. the checkpoint is entirely taken). This is quite a natural approach when thinking that checkpointing instructions have been always charged to the CPU (1).

With semi-asynchronous checkpointing, presented in this dissertation for the context of non-incremental methods, checkpoints of an LP state are taken in real concurrency with the execution of other simulation specific operations (e.g. event list update, event execution and so on), thus reducing the impact of any checkpointing operation on the completion time of the parallel simulation.

In this chapter we present semi-asynchronous checkpointing passing through the description of the model of a system capable to support such a checkpointing mode and the identification of requirements for the effectiveness of this mode. Core issues specifically related to concurrency between checkpointing and other simulation specific operations are identified, and a notion of re-synchronization is introduced to cope with these issues.

1 An exception to charging the checkpointing instructions to the CPU is the Rollback-Chip solution already discussed in Section 2.1.2 of Chapter 2. However, also the Rollback-Chip is based on the synchronous mode for taking checkpoints since the simulation application is suspended until checkpointing actions associated with the MARK command are completely executed.
3.1 System Model

We schematize the data structures associated with the simulation application as partitioned into:

**State Buffers (SB).** This partition contains the buffers storing the states of the LPs hosted by the machine. Denoting with $LP_j$ the $j$-th LP on the machine, we use the notation $sb_j$ to indicate the buffer within SB associated with $LP_j$.

**Checkpoint Stacks (CS).** This partition contains all the stacks of checkpoints associated with the LPs hosted by the machine. We use the notation $stack_j$ to indicate the stack within CS associated with $LP_j$.

**Other Data Structures (ODS).** This partition contains all the remaining data structures kept by the simulator, such as the event lists of the LPs hosted by the machine.

As pointed out, semi-asynchronous checkpointing lies on the idea of real concurrency between the execution of checkpointing and other simulation specific operations. To achieve this, the checkpointing operation must be charged to a device distinct from the CPU. We assume the presence of such a device, which we denote as D. This device has the ability to copy data from SB to CS, so as to perform data transfer of the LP states into the corresponding stacks of checkpoints.

As shown in Figure 3.1, the simulation code executed by the CPU accesses all the three partitions SB, CS and ODS. For example, it copies data from $stack_j$ to $sb_j$ (i.e. from CS to SB) while executing a state recovery procedure in case of rollback of $LP_j$, or accesses ODS while updating the event lists of the LPs. Instead, D accesses exclusively the two partitions SB and CS, copying data from SB to CS (i.e. from $sb_j$ to $stack_j$).

We do not assume D as a special purpose device, therefore, it might perform a set of other user/system level tasks.

For simplicity, we assume that D can handle at most one data transfer associated with checkpointing at a time, therefore no LP is allowed to issue a checkpoint request to D until a previously issued request (by whichever LP) is still being handled. Although this assumption might be theoretically relaxed, we will show that for the specific hardware platform underlying the implementation of semi-asynchronous checkpointing we will present in Chapter 4, the management of a single checkpointing operation at a time is mandatory in practice to respect the effectiveness requirements for the semi-asynchronous mode identified in the following section.
3.2 Effectiveness Requirements

As already explained, the objective of using the device D to support the checkpointing protocol (i.e. data transfer associated with checkpointing) is to allow the CPU to carry out other simulation specific operations while checkpointing is in progress. A basic requirement for the effectiveness of this approach is that, while transferring data due to checkpointing, the device D produces in practice negligible interference on CPU activities. In other words, the execution speed of those activities must not suffer from the activation of checkpointing activities on the device D. We call this requirement as Device Decoupling. If this requirement were not satisfied, charging the checkpointing protocol to the device D might slow down the execution speed of any application running on top of the system, including the simulation application itself, which might, in turn, eliminate any benefit derived from semi-asynchronous checkpointing.

Another point is related to the execution of tasks, distinct from checkpointing, carried out by the device D. If D performs user/system level performance critical tasks (we recall that D is not assumed to be a special purpose device), then charging the checkpointing protocol to the device D must be done in a way not to produce significant interference on the execution of these tasks. We call this requirement as Non-Intrusiveness. If this requirement were not satisfied, any application running on top of the system and exploiting the critical tasks performed by the device D (possibly including the simulation application itself), might exhibit unacceptable performance degradation.
3.3 Data Consistency and Device Contention Issues

Each entry into stack\( j \) maintains a snapshot of the state of LP\( j \) at a given simulation time. Upon the execution of a simulation event, the simulation clock of LP\( j \), i.e. its LVT, moves to the event timestamp. Also, changes of the state can occur due to updates issued on state variables while executing the event itself. A sufficient condition for the consistency of any snapshot maintained into stack\( j \) (i.e. the consistency of any checkpoint for LP\( j \)) is that a complete copy of sb\( j \) into stack\( j \) is performed before LP\( j \) is scheduled for the execution of a new simulation event. When the data copy from sb\( j \) to stack\( j \) is charged to D, there exists the possibility that LP\( j \) is scheduled for event execution while D is still copying the content of sb\( j \) into stack\( j \) (due to a previously activated checkpointing operation for LP\( j \)). If no precaution is taken, the corresponding entry into stack\( j \) might result in an incorrect snapshot of the state of LP\( j \).

In addition, the simulation software run by the CPU might issue a checkpointing request to the device D while the last issued one is still being handled. The new request would originate contention on the device D which cannot be managed since, according to the system model, a single checkpointing request at a time can be handled by D.

Both data consistency and device contention issues are tackled through the notion of re-synchronization, to be explained in the following section.

3.4 Re-Synchronization

Re-synchronization is what makes the checkpointing mode we propose semi-asynchronous, instead of fully asynchronous. A re-synchronization point is characterized by specific actions that we call *semantic* of re-synchronization. They describe what must happen at both the CPU side and the side of the device D in order to allow data consistency to be maintained and to avoid device contention.

Below we discuss four distinct re-synchronization semantics, namely Application Freezing (AF), Checkpoint Abort (CA), Conditional Checkpoint Abort (CCA) and Minimum Cost (MC), pointing out both advantages and potential drawbacks of each semantic.

Before entering the description of re-synchronization semantics, we provide indications on how to use re-synchronization within a classical optimistic simulation engine structure, like the one reported in Figure 3.2. When the checkpointing protocol is charged to the device D, a first re-synchronization point must be located just after the LP scheduling operation in line 3. This is done in order to ensure data consistency by avoiding that the LP scheduled for
## 3.4. RE-SYNCHRONIZATION

```plaintext
1 while(not end)
2  <receive messages/antismessages and update event lists>;
3  sched_LP = <schedule next LP>;
4  if (rollback required for sched_LP)
5     <execute rollback for sched_LP>;
6  <event execution for sched_LP>;
7  <send event notification messages>;
8  if (checkpoint required for sched_LP)
9     <checkpoint for sched_LP>;
10     <advance GVT and collect fossils>;
```

Figure 3.2: A Classical Optimistic Engine Structure.

execution, namely `sched_LP`, issues updates on its state while the state itself is still being transferred into the stack of checkpoints by the device D (otherwise those updates might ultimately result in a non-consistent checkpoint for that LP). Re-synchronization in this point of the engine must be activated only in case `sched_LP` is associated with the last issued semi-asynchronous checkpointing operation. This is because updates issued by `sched_LP` on its state has no impact on the consistency of an ongoing checkpointing operation for a distinct LP.

A second re-synchronization point must be located just before activating a checkpointing operation in line 9 in order to avoid contention on the device D. Specifically, this re-synchronization point prevents issuing a new checkpoint request to the device D in case the last issued one is still being handled. Re-synchronization in this point must be activated only in case the test in line 8 results true (i.e. in case taking a checkpoint is required for `sched_LP`) whose result depends on the adopted policy for selecting when to issue checkpoint requests. As an example, a semi-asynchronous checkpointing operation for a specific LP might be activated each \( \chi \) event executions, in a fashion similar to periodic checkpointing based on synchronous checkpoints, already explained in Section 2.1.1 of Chapter 2.

On the basis of previous considerations, the engine structure should be modified as shown in Figure 3.3. The re-synchronization points appear in line 4 and in line 10. Also, in line 11, the issue of a semi-asynchronous checkpoint request replaces the classical synchronous invocation of the checkpointing protocol (see line 9 of the original engine in Figure 3.2).

We now proceed introducing the re-synchronization semantics.

### 3.4.1 Application Freezing Semantic (\( \mathcal{AF} \))

\( \mathcal{AF} \) is a straightforward semantic for re-synchronization which temporarily suspends (i.e. temporarily freezes) the execution of simulation operations dis-
Figure 3.3: Modified Engine Structure Embedding Semi-Asynchronous Checkpointing.

tinct from checkpointing, carried out by the CPU, until the last activated semi-asynchronous checkpointing operation is completed by the device D. With this semantic, neither sched LP is allowed to update its state while the state itself is being checkpointed (this maintains data consistency), nor it can issue a semi-asynchronous checkpointing request until the last issued one is completely handled by the device D (this avoids device contention).

Using this semantic, if no real freezing occurs whenever re-synchronization points are reached, semi-asynchronous checkpointing effectively removes the checkpointing cost from the completion time of the simulation application. On the other hand, in case of frequent, non-minimal freezing periods, the application might suffer from non-negligible checkpointing cost, with a reduction of the effectiveness of the semi-asynchronous mode.

3.4.2 Checkpoint Abort Semantic (CA)

CA is a semantic opposite to AF. With CA simulation operations distinct from checkpointing are never frozen at any re-synchronization point. Instead, the effect of CA is to abort any not yet completed semi-asynchronous checkpointing operation. Data consistency is maintained since, in case sched LP is associated with a not yet completed semi-asynchronous checkpointing operation, the operation itself is considered as if it had never been activated. Therefore, updates issued by sched LP on its state will never ultimately result in a non-consistent checkpoint since the situation looks like as if no checkpoint at all was ever scheduled for the state value prior to the updates. Analogously, no device contention ever occurs since any ongoing checkpointing operation is interrupted (i.e. taken away from the device D) before any new checkpointing operation is activated.
With this semantic, the simulation application experiences no checkpointing cost at all since, unlike \( \mathcal{AF} \), no application freezing occurs at any re-synchronization point (i.e. the application is never suspended to wait for the completion of an ongoing semi-asynchronous checkpointing operation). However, under certain circumstances, we might incur rollback thrashing. More precisely, in case very frequent checkpoint aborts are experienced, the likelihood of very high state recovery cost increases due to the increase of the average distance between consecutive “committed” checkpoints for the same LP, which might lead to excessively long coasting forwards. In addition, as already discussed in Section 2.1.1 of Chapter 2, excessively long coasting forwards might even originate an increase in the amount of rollback, contributing to rollback thrashing.

3.4.3 Conditional Checkpoint Abort Semantic (CCA)

CCA is a semantic that merges the advantages of both \( \mathcal{AF} \) and \( \mathcal{CA} \). Like \( \mathcal{CA} \) it reduces the checkpointing cost experienced at the application level due to freezing. On the other hand, like \( \mathcal{AF} \) it avoids rollback thrashing by allowing to control the amount of committed semi-asynchronous checkpointing operations for any LP.

Under \( \mathcal{CCA} \) the simulation application is frozen, at a given re-synchronization point, only in case at least a threshold fraction, “threshold” for short, of the lastly activated semi-asynchronous checkpointing operation has already been carried out by the device D. In the opposite case, the ongoing checkpointing operation is aborted and the simulation application is allowed to proceed, with no freezing at all.

When the threshold value is set to zero, \( \mathcal{CCA} \) boils down to \( \mathcal{AF} \) since, upon the occurrence of re-synchronization, the application is frozen until the completion of the ongoing checkpointing operation independently of the advancement status of the operation itself. On the other hand, when the threshold value is set to one, \( \mathcal{CCA} \) boils down to \( \mathcal{CA} \) since re-synchronization has the effect to abort any not yet completed checkpointing operation. Therefore, \( \mathcal{CCA} \) can simulate both the previously described semantics. Also, (dynamically) controlling the threshold value is expected to allow adequate tradeoffs between the checkpointing cost experienced at the application level in the form of freezing and the state recovery cost determined on the basis of the distance between consecutive committed checkpoints at the same LP. This allows the effects of re-synchronization to be tailored to proper execution dynamics of the overlying simulation application.

We note however that \( \mathcal{CCA} \) needs the ability to efficiently/effectively track the advancement status of an ongoing checkpointing operation (\(^2\)). This re-

\(^2\)Efficiency means that advancement tracking must be performed at low cost. Effectiveness
quirement disappears for the case of \( \mathcal{CA} \) and \( \mathcal{AF} \), since these two semantics only need the ability to abort and/or to track the completion of a checkpointing operation. If for any reason efficient/effective tracking of the advancement status is inviable in practice on the used instance of the previously assumed system model, a combined use of \( \mathcal{AF} \) and \( \mathcal{CA} \) represents a practical alternative to \( \mathcal{CC}A \).

3.4.4 Minimum Cost Semantic (\( \mathcal{MC} \))

Minimum Cost (\( \mathcal{MC} \)) is the most advanced semantic presented in this dissertation. It takes the decision on whether to commit/abort an ongoing semi-asynchronous checkpointing operation upon re-synchronization on the basis of a cost model that associates with any re-synchronization point a checkpointing-recovery overhead. The checkpointing overhead is evaluated as the expected residual completion latency of the ongoing checkpointing operation \(^3\). The recovery overhead is the time penalty associated with commit/abort of that checkpoint. The convenience of checkpoint commit/abort is then established solving the model and the more convenient action is executed. We shall present this semantic by first introducing the cost model.

The Cost Model

Let us consider a portion of the evolution of an LP in simulation time as shown in Figure 3.4. Black circles represent events already executed by the LP, empty

\(^3\)Estimation of the expected residual completion latency actually needs knowledge on the advancement status of the semi-asynchronous checkpointing operation, thus \( \mathcal{MC} \) actually exhibits the same requirement as \( CC\mathcal{A} \) as respect to tracking the advancement status of the operation.
circles represent events not yet executed, and labeled boxes represent state values at given points in the simulation time that is, those points corresponding to event timestamps. A state value is associated with each executed event. This value results from updates issued by the LP on state variables while executing that event. In our example, the state value $X$ results from updates issued on the state value $Y$ due to the execution of the event $e$. We associate with each state value $X$ traversed by the LP a probability value, namely $P(X)$, which is the probability that a future rollback needs state recovery exactly to the state value $X$.

Suppose we are currently taking a snapshot of the state value $X$ using the semi-asynchronous execution mode of the checkpointing protocol (i.e. the device D is transferring data from $s_{bj}$ into stack$_j$) and re-synchronization occurs either because the LP has been re-scheduled for the execution of its next event, namely $e'$ in Figure 3.4, or because some other LP needs to issue a semi-asynchronous checkpoint request. At this point we must establish the convenience of committing/aborting the semi-asynchronous checkpointing operation currently involving the state value $X$. We recall that commit causes temporarily freezing of simulation software run at the host side until the completion of the checkpointing operation. Instead, abort leads to skipping that checkpoint, with possible impact on the state recovery latency due to the reduced amount of available (committed) checkpoints.

Denoting with $\Delta_{\text{ckpt}}^{\text{completion}}$ the expected residual completion latency for the checkpointing operation at re-synchronization occurrence, and with $\Delta_{\text{ckpt}}^{\text{interrupt}}$ the average latency to handle checkpoint abort, the checkpointing overhead $OH_{\text{ckpt}}$ associated with re-synchronization can be expressed as

$$OH_{\text{ckpt}} = \begin{cases} \Delta_{\text{ckpt}}^{\text{completion}} & \text{commit case} \\ \Delta_{\text{ckpt}}^{\text{interrupt}} & \text{abort case} \end{cases}$$

Actually, $\Delta_{\text{ckpt}}^{\text{completion}}$ corresponds to the freezing interval on CPU activities in case a commit decision is taken for the ongoing checkpointing operation. $\Delta_{\text{ckpt}}^{\text{interrupt}}$ represents instead the latency for notifying to the device D that the data transfer associated with the ongoing operation must be interrupted.

Taking the checkpoint of the state value $X$ does not affect the recovery time to any state value $Z$ preceding $X$. This is because the state recovery time to $Z$ depends only on the position of the latest checkpoint preceding (or coinciding with) $Z$, and on the granularity of the intermediate events, if any, from that checkpoint to the state value $Z$. Specifically, if $Z$ has not been checkpointed and must be recovered due to rollback, then coasting forward (i.e. event execution replay) is needed from the latest checkpoint preceding $Z$, to $Z$. On the other hand, if $Z$ has been checkpointed, then state recovery only entails reloading the checkpointed value into the LP state buffer. In
both cases, the time for the state recovery operation is not affected by the commit/abort of the checkpointing operation involving the state value \( X \).

By the previous discussion, the recovery overhead \( OH_{\text{recovery}} \) associated with re-synchronization can be evaluated as the additional expected recovery cost due to recovery of the state value \( X \) in case of rollback. Such a cost varies depending on whether the ongoing checkpointing operation involving \( X \) is committed or aborted. Specifically, in case the checkpointing operation is committed, then the state recovery overhead due to a rollback to \( X \) is the time to reload the checkpointed state value \( X \) into the LP state buffer. Otherwise, coasting forward is required. Denoting with (i) \( EV(X) \) the set of all the events that move the LP from the latest checkpointed state value preceding \( X \), to \( X \), (ii) \( \Delta_e \) the granularity (execution time) of the event \( e \in EV(X) \), and (iii) \( \Delta_{\text{reload}} \) the time to reload a checkpointed state value into the checkpoint buffer of the LP, then we can express \( OH_{\text{recovery}} \) as

\[
OH_{\text{recovery}} = \begin{cases} 
P(X) \Delta_{\text{reload}} & \text{commit case} \\
\frac{P(X) \Delta_{\text{reload}} + \sum_{e \in EV(X)} \Delta_e}{P(X) + \Delta_{\text{reload}}} & \text{abort case}
\end{cases}
\] (3.2)

Expression (3.2) states that if the checkpointing operation involving \( X \) is eventually committed, then in case of state recovery to \( X \) (this happens with probability \( P(X) \)) the recovery overhead consists only of the time \( \Delta_{\text{reload}} \) to reload the checkpointed state value \( X \) into the LP state buffer. Otherwise, it consists of the time to reload the latest checkpointed state value preceding \( X \) plus the time to replay all the events in \( EV(X) \) that is, the coasting-forward time.

Summing contributions in (3.1) and (3.2) we get the following expression for the whole checkpointing-recovery overhead associated with the re-synchronization point

\[
OH_{\text{ckpt}} + OH_{\text{recovery}} = \begin{cases} 
\Delta_{\text{ckpt}}^{\text{complete}} + P(X) \Delta_{\text{reload}} & \text{commit case} \\
\Delta_{\text{ckpt}}^{\text{interrupt}} + P(X) \Delta_{\text{reload}} + \sum_{e \in EV(X)} \Delta_e & \text{abort case}
\end{cases}
\] (3.3)

We note that expressions for \( OH_{\text{recovery}} \) and \( OH_{\text{ckpt}} \) have been derived by implicitly assuming that \( P(X) \) does not change depending on whether the ongoing checkpointing operation involving the state value \( X \) is committed or aborted. In other words, we have assumed that committing or aborting the
checkpoint will result in no perceptible change in the rollback behavior. As respect to this point, we note that committing or aborting a checkpoint at a specific re-synchronization point will ultimately result in slightly shorter or longer average distance between checkpoints for the LP, which is commonly assumed to have in practice no significant effect on the rollback behavior by models underlying checkpointing modes based on the CPU charged (synchronous) execution mode of the checkpointing protocol [48, 67, 80, 83]. As respect to this assumption, [71] shows that, although relevant variations of the rollback behavior may be noted for very large increments of the distance between checkpoints, they are less likely to be observed for slight variations of that distance. Therefore, the assumption itself does not appear as unrealistic.

**Definition of $\mathcal{MC}$ Re-synchronization**

The $\mathcal{MC}$ semantic for re-synchronization should commit/abort the ongoing checkpointing operation in order to minimize the checkpointing-recovery overhead as expressed in (3.3). However, we note that in case the last activated semi-asynchronous checkpointing operation has been already completed at re-synchronization occurrence, an abort decision does not make sense. This is reflected by the checkpointing-recovery cost model since an already completed checkpoint actually means a null value for $\Delta_{\text{ckpt}}$, which in turn leads the commit decision to always minimize the checkpointing-recovery overhead due to the fact that

$$P(X)\Delta_{\text{reload}} \leq \Delta_{\text{interrupt}} + P(X)[\Delta_{\text{reload}} + \sum_{e \in EV(X)} \Delta_e]$$

(3.4)

Previous observation allows us to formally define $\mathcal{MC}$ in a way that allows sometimes to skip the evaluation of the cost model in (3.3) since we know in advance that no checkpoint abort needs to be performed at the re-synchronization point (i.e. when the semi-asynchronous checkpointing operation has already been completed). Denoting with $x$ and $y$ the values obtained by evaluating the cost model in (3.3) in case of commit and abort, respectively, our definition of $\mathcal{MC}$ is reported in Figure 3.5. From a practical view point, the test in line 1 and the statement in line 2 allow a reduction of the overhead associated with $\mathcal{MC}$ since, in case the test is verified and the statement is executed, no calculation of $\text{value} = x - y$ must be performed (i.e. the cost model does not need to be solved upon re-synchronization).

Solving the checkpointing-recovery cost model, i.e. computing the values $x$ and $y$ in line 3, requires knowledge of several parameter values. Some of these parameters appear in the literature in performance models for classical CPU charged (synchronous) checkpointing (i.e. $\Delta_c$, $\Delta_{\text{reload}}$ and $P(X)$), and a set of solutions have already been proposed for determining their values.
1. if last activated checkpointing operation already completed
2. <return COMMIT>
3. <compute value = x - y>
4. if value ≤ 0
5. <wait for checkpoint completion>
6. <return COMMIT>
7. else
8. <interrupt the ongoing checkpointing operation>
9. <return ABORT>

Figure 3.5: The \(MC\) Re-synchronization Semantic.

refer to these parameters as classical and we briefly recall here some of those solutions for the reader’s convenience.

Determination of non-classical parameters proper of semi-asynchronous checkpointing (i.e. \(\Delta_{\text{complete}}\) and \(\Delta_{\text{interrupt}}\)) is dependent on the specific nature of the device \(D\), and therefore on the specific implementation of semi-asynchronous checkpointing. An example will be presented in Chapter 4, where the non-classical parameters will be determined for the specific implementation presented in this dissertation.

**Determination of Classical Parameter Values**

Classical parameters appearing in models for synchronous checkpointing are (i) the event execution time, namely \(\Delta_e\), (ii) the time \(\Delta_{\text{cloud}}\) to reload a checkpointed state value into the LP state buffer [20, 48, 67, 76, 80, 83], and also (iii) the probability of recovery to a specific state value \(X\), namely \(P(X)\) [76]. The first parameter has been traditionally approximated with an expected event granularity value computed over the set of granularity values characterizing the specific simulation application [20, 48, 67, 76, 80]. Its computation is performed either off-line or during the early phase of the simulation application execution. This approach is typically not expensive and results effective in case of low variance in the execution time of distinct events. However, for applications with large variance in the event granularity, it might be inadequate since the expected value might not be a reliable indicator for the granularity of a given event. For such simulation applications there exist recent solutions that point out how to measure on-line the granularity of each executed event with negligible overhead by using either fast access real time clocks or accounting instructions embedded within the application code [76]. These approaches could be straightforwardly used for keeping track of the granularity \(\Delta_e\) of each already executed event \(e\) in the set \(EV(X)\) characterizing our checkpointing-
3.4. RE-SYNCHRONIZATION

![Diagram showing state value X and simulation time with events e and e', interval I(X)]

Figure 3.6: The Interval $I(X)$ Associated with $X$.

recovery cost model.

For what concerns $\Delta_{\text{reloads}}$, we note that computation of $value = x - y$ in line 3 of the algorithm in Figure 3.5 actually does not really need knowledge of this parameter. In other words, solving the cost model as the difference between $x$ and $y$ allows us to ignore that parameter value.

As respect to $P(X)$, approaches to the estimation of this parameter can be found in [29, 76]. They are based on associating with any state value $X$ a left open interval of simulation time $I(X)$ evaluated as the difference between the timestamp of the event that moves the LP from the state value $X$ to a successive value, and the timestamp of the event that moved the LP state just to the value $X$. An example is shown in Figure 3.6 where the interval associated with the state value $X$ can be expressed as

$$I(X) = (\text{timestamp}(e), \text{timestamp}(e'))$$

(3.5)

where $e$ is the event moving the state to the value $X$ and $e'$ is the next event to be executed by the LP. Since $P(X)$ corresponds to the probability that a timestamp order violation will eventually occur in the interval $I(X)$ $^{(4)}$, these solutions estimate $P(X)$ exploiting the length of the interval $I(X)$ and monitoring the relative frequency of timestamp order violation occurrences in simulation time intervals of a given length. In other words, these solutions maintain a histogram for the relative frequency of state recovery in simulation time intervals of pre-specified lengths. How to determine the length of those intervals in order to get balanced observations is also discussed [29].

As a last point related to the estimation of $P(X)$, we note that if the histogram maintains data related to a unique right open interval $[0, \infty)$, then $P(X)$ is simply estimated as the ratio between the total number of rollbacks occurring at the LP and the total number of event executions. This ratio is commonly referred to as the rollback frequency of the LP. Such an approach

---

$^{(4)}$After $e'$ is executed, the violation in the interval $I(X)$ can be caused by the scheduling for that LP of an event $e''$ with $\text{timestamp}(e'') \in I(X)$, or even by the arrival of the antimessage that cancels $e'$. 

to the estimation of \( P(X) \) favors simplicity and keeps the overhead low at the expense of a reduced exploitation of the real rollback pattern of the LP. As we will show in Section 5.1 of Chapter 5, \( \mathcal{MC} \) reveals effective even adopting this simplified approach for the estimation of \( P(X) \). This is an expected behavior when thinking that \( \mathcal{MC} \) maps the values of a continuous function of \( P(X) \), namely the difference between the two checkpointing-recovery costs in case of commit/abort of the checkpointing operation, into a boolean domain, namely a commit/abort decision. As already pointed out in [76] while discussing a model for synchronous checkpointing, this kind of mapping removes the effects of limited quality in the estimation process unless the difference between the real and the estimated values of \( P(X) \) over-steps a given threshold. This should contribute to robustness of \( \mathcal{MC} \) even in case of simple estimation methods of \( P(X) \).

### 3.4.5 Comparative Discussion Between \( \mathcal{CCA} \) and \( \mathcal{MC} \)

Although both \( \mathcal{CCA} \) and \( \mathcal{MC} \) are aimed at improving performance by committing/aborting an ongoing checkpointing operation at the re-synchronization point, they are totally different in nature. In \( \mathcal{CCA} \) the criterion for taking the commit/abort decision, namely the threshold percentage for the ongoing checkpointing operation, is predetermined upon re-synchronization occurrence. Instead, \( \mathcal{MC} \) is based on a criterion which is not predetermined, namely the result of the evaluation of the checkpointing-recovery cost model upon re-synchronization occurrence.

Any predetermined criterion for committing/aborting an ongoing checkpointing operation is inherently less capable of providing higher performance benefits as compared to a non-predetermined one. This is because a predetermined criterion provides less flexible decisions, which might not be definitely adequate for a specific re-synchronization point. As we will show by the experimental results in Chapter 5, the non-predetermined approach allows \( \mathcal{MC} \) to outperform \( \mathcal{CCA} \) when the likelihood for the last activated semi-asynchronous checkpointing operation to be not yet completed upon re-synchronization is non-minimal, i.e. when the criterion plays a relevant role to determine commitment (or uncommitment) of ongoing checkpointing operations\(^5\). Conversely, when we frequently find an already completed checkpointing operation upon re-synchronization, (as it will be shown in Chapter 5 this is typical of simulation with small size of the LP state), we expect \( \mathcal{CCA} \) and \( \mathcal{MC} \) to provide similar performance due to the reduced impact of the selected criterion on commitment/uncommitment of checkpoints.

\(^5\)As discussed in Section 3.1, an already completed checkpointing operation means in practice a committed checkpoint. Therefore no abort decision can be taken by the re-synchronization function.
However, $\mathcal{MC}$ may not always be the straightforward choice. Specifically, such a semantic might reduce transparency at the application programmer level since the evaluation of the underlying cost model requires measuring/estimation of a set of application-level parameters. This is not required by $\mathcal{CCA}$ since, as we will show in Section 5.1.2.6 of Chapter 5, the determination of the threshold value to be used can rely on simple heuristics based on monitoring a single performance parameter. Should that issues arise, $\mathcal{CCA}$ qualifies as a reasonable candidate to substitute $\mathcal{MC}$.

### 3.5 Remarks

The concept of checkpoint abort, characterizing the $\mathcal{CA}$, $\mathcal{CCA}$ and $\mathcal{MC}$ semantics, introduces a skipping effect in taking some checkpoints. As already pointed out in Chapter 2.1.1, checkpoint skipping, namely sparse checkpointing, is a classical solution for reducing the checkpointing overhead in case of non-incremental methods based on synchronous (CPU charged) execution of the checkpointing protocol. However, in those methods, skipping is achieved by completely avoiding the activation of the checkpointing operation at specific execution points, which is rather different from aborting a checkpointing operation after the operation itself has already been activated. More precisely, the notion of checkpoint abort underlying $\mathcal{CA}$, $\mathcal{CCA}$ and $\mathcal{MC}$ actually introduces a kind of a-posteriori checkpoint skipping, where “a-posteriori” means “after” the activation of the checkpointing operation itself. This is orthogonal to the classical a-priori checkpoint skipping performed by existing sparse state saving methods. Therefore the two skipping approaches can be combined with each other. As respect to this point, if we look at the structure of the simulation engine embedding semi-asynchronous checkpointing in Figure 3.3, we note that the engine exhibits a test in line 9 to verify whether a checkpointing operation must be activated for the lastly scheduled LP, namely $\text{sched\_LP}$. In the negative case, a-priori checkpoint skipping is performed independently of the fact that the execution mode of the checkpointing protocol is semi-asynchronous. Insights on the effects of combining a-priori and a-posteriori checkpoint skipping will be provided in the experimental analysis reported in Section 5.1.3 of Chapter 5.

As a last observation, we recall that classical sparse checkpointing based on a-priori skipping of checkpoints has the advantage of keeping low the memory usage due to the reduced amount of recorded state information per event execution, which might exhibit benefits when memory is a critical resource. Given the possibility to adopt a-priori skipping in combination with the semi-asynchronous execution mode of the checkpointing protocol, such an advantage can be maintained while using semi-asynchronous checkpointing. Overall,
semi-asynchronous checkpointing exhibits the potential to reduce the cost of any single checkpointing operation while still allowing the possibility to reduce memory usage whenever needed.
Chapter 4

An Implementation for Myrinet Clusters

“The map is not the territory.”
Alfred Korzybski, Science and sanity

This chapter describes the implementation of a C library capable to support the semi-asynchronous execution mode of the checkpointing protocol on clusters connected by a commercial high-speed, low-latency interconnection network known as Myrinet. Actually, this library exploits DMA capabilities proper of the myrinet network hardware to support the data transfer associated with checkpointing. Specifically, in our perspective, the device D coincides with a myrinet network card. As we will show, usage of the myrinet network card as the device D actually respects the Device Decoupling requirement identified in Section 3.2 of Chapter 3. In addition, proper software design will also allow to respect the Non-Intrusiveness requirement identified in the same section, thus making checkpointing not to interfere with message transfer operations, namely the performance critical task supported by the network card. Beyond the presence of other applications which may suffer from performance decrease in the communication system, we recall that in the context of optimistic PDES, communication is typically a performance critical task since, as shown in [12], any increase in the delivery delay of messages/antimessages might mean higher likelihood of incorrect computation, i.e. likelihood of higher amount of rollback.

The library presented here, that we will refer to as Checkpointing-and-Communication Library (CCL), is an integrated software offering both checkpointing and low latency message delivery functionalities. A description of the myrinet hardware and an overview of typical message passing layers for myrinet are mandatory for the comprehension of the CCL structure. We will provide such preliminary information in Section 4.1. Subsequently, Section 4.2
presents a detailed description of the implementation of semi-asynchronous
checkpointing, together with message passing functionalities. Finally, Section
4.3 describes how parameters proper of the implementation should be tuned
to achieve the best performance.

As a last observation, we remark that CCL has been developed for LINUX,
kernel version 2.0.32, assuming an underlying hardware architecture employing
snooping-cache. However, porting on newer kernel releases and/or on hard-
ware architectures that need explicit invalidation to maintain cache coherency,
does not require any relevant programming effort. Also, the same algorithms
used in the implementation can be re-used in case of a re-implementation for
other UNIX-like or Windows systems.

4.1 Myrinet Overview

4.1.1 Hardware Platform

Myrinet networks [59] (produced by Myricom [58]) are off-the-shelf, cost-
effective, high-performance networks based on cut-through switching technol-
yogy, widely used by the parallel computing community to interconnect work-
stations and/or PCs.

Its adoption is mainly due to:

☐ Low latency, obtained with cut-through technology, crossbar switching
   and Operating System (OS) bypass mechanisms;

☐ High bandwidth (full-duplex 2+2 Gigabit/second data rate);

☐ Networks that can scale to tens of thousands of hosts;

![Myrinet Network Architecture](image-url)

Figure 4.1: Myrinet Network Architecture.
A myrinet network is mainly composed by three components, as schematized in Figure 4.1:

1. **Myrinet Network Cards**, which are the interfaces between the myrinet networks and the machines interconnected by it.

2. **Myrinet Switches**, which are crossbar switches allowing the interconnection of a scalable number of machines.

3. **Myrinet Links**, which are cable or optical fiber links designed for high-bandwidth and low-latency communication between two separate nodes (switches and/or network cards) of the network.

CCL does not interfere at all with the network itself (i.e. switches or links). However it exploits the DMA capabilities of the network card, therefore a more detailed description of the myrinet network card follows.

**Network Card in Detail.** The version of CCL presented in this dissertation has been designed for the M2M-PCI32C myrinet card (see Figure 4.2), based on the LANai 4 chip [60]. This network card is a programmable communication device consisting of:

(A) An internal bus, namely LBUS (Local BUS), clocked at twice the chip-clock speed.

(B) A programmable RISC processor connected to the LBUS, which we will refer to as LANai processor.

(C) A RAM bank of 1 Mbyte (LANai internal memory), connected to the LBUS, which is used for storing both data and the driver run by the LANai processor, typically called control program. This memory can be mapped into the memory address space of the host. Also, host access to the LANai internal memory takes place through a PCI bridge.

(D) A packet interface between the myrinet switch and the LANai chip, accessible by the LANai processor.

(E) Three DMA engines used respectively for:

1. Packet-interface/internal-memory transfer operations (Receive DMA);
2. Internal-memory/packet-interface transfer operations (Send DMA);
3. Internal-memory/host-memory transfer (or vice-versa) operations (EBUS DMA, namely External Bus DMA). Internal-memory/host-memory transfer (or vice-versa) also takes place through the PCI bridge.
The LANai processor cannot access host memory directly. Nonetheless, the control program run by the LANai processor can set the EBUS DMA to perform data transfer to/from that memory.

Note that, although the LBUS can perform two data transfers per chip-clock cycle, there are five sources of potentially concurrent transfer requests, namely the host processor (i.e. the CPU), three distinct DMA engines and the RISC processor. Thus conflicting LBUS requests may occur. In case of concurrent requests, the LBUS cycles are allocated according to the following priority order: host and EBUS DMA, Receive DMA, Send DMA, RISC processor.

### 4.1.2 Typical Message Passing Layer

The general architecture of typical fast speed messaging layers for myrinet networks (e.g [62] or [66]) is schematized in Figure 4.3. Similarly to other communication layers an API for communication is exposed by an application level library (i.e. the Software Interface).

However, differently from many other commercial networks, the Software Interface does not rely on software at the Operating System level (i.e. the...
Driver) to perform communication, since basic communication functionalities are not built inside the Driver. The main use of the Driver is simply to correctly reset the myrinet network card and map it into the application address space.

Instead network communication is realized by a direct interaction of the Software Interface with the myrinet network card (controlled by the control program), thus bypassing the OS. By skipping the OS abstraction level the communication latency is reduced.

To summarize, the software modules directly involved in communication are only the Software Interface and the control program. The Driver is only required to initialize the network card (i.e. load the control program on the network card and bring it out of reset) and setup the OS bypass mechanisms.

Typically message passing functionalities in a myrinet layer are realized as follows:

- **Receive Operation**

  Messages incoming from the network are temporarily buffered into the LANai internal memory (data transfer between the packet interface and the internal memory takes place through the Receive DMA) and then transferred into a *receive queue*, located onto host memory, through the EBUS DMA (see the directed dashed line in Figure 4.2). Once transferred into the receive queue, any message is received by the application program very efficiently by simply performing a `memcpy()` operation of the message content into a proper buffer in the application address space. Given that the message is already in the host memory, this operation
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>While (1)</td>
</tr>
<tr>
<td>2.</td>
<td>if (message needs to be sent) &lt;activate send DMA&gt;;</td>
</tr>
<tr>
<td>3.</td>
<td>if (message needs to be received) &lt;activate receive DMA&gt;;</td>
</tr>
<tr>
<td>4.</td>
<td>if (EBUS DMA not busy)</td>
</tr>
<tr>
<td>5.</td>
<td>if (block-DMA needed) &lt;activate block-DMA&gt;;</td>
</tr>
</tbody>
</table>

Figure 4.4: High Level Structure of the Control Program.

does not involve access to the PCI bridge, thus keeping at a minimum the overhead of a message receipt operation. A classical optimization called "block-DMA" is used to transfer incoming messages from the LANai internal memory to the host memory. This optimization allows incoming messages stored in contiguous message slots of the LANai internal memory to be transferred using a single EBUS DMA operation.

- **Send Operation**

  A send operation issued by the application copies the message content directly into the LANai internal memory, without an intermediate copy into an OS-level buffer. This technique is also referred to as "zero-copy" send. Then the message is transferred onto the network through the Send DMA. This optimization allows keeping the message delivery latency at a minimum by avoiding intermediate buffering at the sender side.

  The responsibility to program the three DMA engines anytime there is the need for a given data transfer operation pertains to the control program run by the LANai processor, whose typical high level structure can be schematized as in Figure 4.4.

### 4.2 An Integrated Library for Checkpointing and Communication

As already outlined at the beginning of this chapter, we identify the device D described in Chapter 3 with the myrinet network card. This identification is justified, since the network card meets the basic assumption in Section 3.1 of Chapter 3 discussed while presenting the system model for supporting semi-asynchronous checkpointing. Specifically, the myrinet network card can copy the content of any host-memory buffer (which includes a generic $sb_j$ in SB) into any other host-memory buffer (e.g. a generic $stack_j$ in CS).

This operation can be accomplished by the means of the EBUS DMA, which although not capable to perform such a transfer directly, can use the LANai internal memory as a temporary buffer for the data transfer operation. This perspective is shown in Figure 4.5. In other words, a checkpointing
operation issued at the application level actually means requesting the LANai processor to program the EBUS DMA for the necessary data transfers.

Therefore, we have developed an integrated layer supporting both communication and checkpointing functionalities. As already outlined, the name of the implemented software is Checkpointing-and-Communication Library (CCL).

The main practical problem with the suggested solution is that the simulation application typically manages only virtual addresses, while data transfer through the EBUS DMA requires knowledge of the physical addresses of both the LP current state buffer ($sb_j$) and the checkpoint buffer into the stack (a specific location into $stack_j$). This is because the EBUS DMA supports data transfer operations based exclusively on physical addresses.

A second problem is that, even if the translation of virtual addresses into physical addresses were readily available, the OS might modify this mapping while the DMA is still working (for instance in case of a page-swapout). Therefore a mechanism to lock ($pin$ $down$) the physical memory must be provided.

How we have tackled those problems is described in this section where the complete structure of CCL is presented.
4.2.1 Software Architecture

The software architecture of CCL, schematized in Figure 4.6, consists of three distinct parts: a Kernel Module (KM), a Software Layer (SL) accessible by the simulation application through an API, and a specialized Extended Control Program (ECP) for the LANai processor. In addition, the standard software offered by Myricom to reset the network card and map it into the application address space is employed [61].

Details on each original part are separately reported in the following paragraphs.

**KM.** KM has been introduced mainly to solve the problems raised above i.e. virtual-to-physical address translation and physical memory locking. Specifically, we have decided to solve both those problems by reserving blocks of physical memory pages for both the current state buffers of the LPs (SB) and their stacks of checkpointed states (CS). Reserved blocks are automatically pinned down and their physical address is known to the kernel. Therefore, KM reserves those blocks on its startup, and records both the physical address of the first page of each block and the block size. In our implementation, semi-asynchronous checkpointing operations will involve data located only on physical memory pages associated with these blocks. As an alternative de-
4.2. AN INTEGRATED LIBRARY FOR CHECKPOINTING AND COMMUNICATION

sign choice, KM might have included a new system call capable to perform memory locking on application request but execution of system calls may slow down the simulation application. On the whole the present design is likely to be more efficient. The current version of KM has been designed for LINUX 2.0.32. Porting the software architecture for semi-asynchronous checkpointing on other OSs involves the use of OS dependent features to implement the current functionality offered by KM.

Reservation of blocks of pages takes place through the function  

\_get\_free\_pages() internal to the kernel. Any reserved page will never be swapped out from the physical memory or manipulated in any way by the operating system. The physical addresses of this memory are reported to any interested application running on the system by adding a new file to the /proc file system. From this information any application which knows the name of that file can map those pages in its virtual address space.

SL. SL offers to the application programmer two APIs, one for communication and the other for (semi-asynchronous) checkpointing.

As in typical myrinet communication layers SL interacts directly with the myrinet network card (and therefore with ECP), bypassing the OS. After the setup phase the myrinet driver and KM are no longer active modules. The only two active modules, interacting with each other are SL and ECP.

As schematized in Figure 4.6, SL can be seen as composed by three separate sub-layers. The lower sub-layer offers communication functionalities (directly accessible by the application through the communication API) and low-level EBUS DMA control functionalities.

The mid-level sub-layer exploits those low-level EBUS DMA functionalities to implement an asynchronous memcpy() function used by the upper sub-layer. It is responsibility of this sub-layer to initialize and maintain a mapping between physical and virtual addresses, so that the virtual addresses passed by the upper sub-layer is correctly converted into physical addresses to be used by the lower sub-layer. This allows maintenance of transparency at the application programmer level, since the programmer does not need to know anything about physical addresses.

The upper sub-layer offers to the application a semi-asynchronous checkpointing API specifically designed for optimistic simulation. The partition of the pinned down memory into blocks corresponding to the abstractions of state buffers and checkpointing stacks entries is carried out at this level. Semi-asynchronous checkpoints are taken by employing the asynchronous memcpy() function made available by the mid-level sub-layer.

Details on the communication and checkpointing APIs are reported in Section 4.2.2 and 4.2.3 respectively.
ECP. The ECP run by the LANai processor is designed in a fashion similar to a common control program so far as communication is concerned. However it has been extended to accommodate functionalities to program the EBUS DMA according to requests coming from the lower sub-layer of SL, with the least possible interference on communication functionalities. A detailed description of ECP is given in Section 4.2.4.

4.2.2 Communication Functionalities

Communication functionalities provided by CCL have been implemented to fully exploit the potential offered by the hardware components on the network card. Therefore the techniques of the previously mentioned classical communication layers have been adopted. Specifically, messages incoming from the network are still temporarily buffered into the LANai internal memory, transferred into the host-memory receive queue through the EBUS DMA, and received by the application program by performing a `memcpy()` operation.

In the same fashion, any send operation issued by the application adopts the “zero-copy” method (i.e. messages are written directly into the LANai internal memory and transferred onto the network through the Send DMA).

The responsibility to program the three DMA engines anytime there is the need for a given data transfer operation pertains to the ECP run by the LANai processor. The same program also handles an acknowledgment mechanism to support reliability of message delivery. In the current implementation, acknowledgments are 4 bytes packets and message re-transmission is performed by the ECP in case acknowledgments do not arrive within a given timeout.

The communication API consists in the following functions, available at the application level.

- int send_msg(int msg_type, int machine_id, void *msg), where
  msg_type defines the type of the message, machine_id defines the destination host for the message, and msg is a pointer to the memory area containing the data to be sent (1); this function passes the data to be transmitted to the network card employing the zero-copy optimization previously described.

- int receive_msg(int msg_type, void *msg, int *machine_id),
  which returns a message of a given type, if any, in the memory area pointed by msg, and also the identifier of the sender host in the memory area pointed by machine_id; this function copies the message content

---

1In the current implementation the maximum amount of bytes that can be sent via a single message is determined at compile time of CCL so that it can be tailored to the requirements of any overlaying application.
from the receive queue to the application address space through a simple 

`memcpy()` call.

Message delivery latencies offered by CCL for specific message sizes are aligned with those offered by other myrinet tailored message passing layers [66]. Information on delivery latency values for the specific message size used in the experimental analysis presented in this dissertation will be provided when describing the experimental environment, at the beginning of Chapter 5.

### 4.2.3 Checkpointing Functionalities

As already explained, in CCL a semi-asynchronous checkpointing operation for \(LP_j\) corresponds to a data transfer from the state buffer \(sb_j\) to an entry the stack of checkpoints \(stack_j\), which is charged to the EBUS DMA. The EBUS DMA uses the LANai internal memory as a temporary buffer for the transfer operation. Temporary buffering is needed since, as already mentioned, the EBUS DMA does not support host memory to host memory data transfer directly. It only supports host memory to LANai internal memory transfer or vice-versa. Anyway, we underline that conventional computer architectures (e.g. IA-32 [42, 43, 44]) are equipped with no hardware component, distinct from the CPU, able to perform host memory to host memory data copy, thus making the intermediate buffering approach the only feasible solution for the semi-asynchronous checkpointing mode.

Using the EBUS DMA on the myrinet card to perform data transfer associated with checkpointing matches the Device Decoupling requirement identified in Section 3.2. Specifically, data transfer performed by the EBUS DMA involves access to the host main memory while the CPU works into the cache memory. In other words, there is no direct interference due to EBUS DMA data transfer on CPU activities. Actually, charging data transfer associated with checkpointing to the EBUS DMA could produce a form of indirect interference due to variations in the execution locality. Specifically, checkpoint stack entries are not referenced by the CPU while taking the checkpoint, therefore they are not loaded into the cache while performing the checkpointing operation. This might have both positive and negative effects. On the positive side, contention on cache entries is reduced while taking the checkpoint. On the negative side, the checkpoint stack entry is not available into the cache for future reference. However, even if the synchronous (i.e. CPU based) mode has the effect of loading any referenced checkpoint stack entry into the cache while taking the checkpoint, the involved cache entries might get overwritten before a new reference to that checkpoint stack entry is made by the simulation program. Therefore, synchronous execution might not favor locality anyway.
Also, a second form of indirect interference due to EBUS DMA data transfer associated with checkpointing is related to the system bus traffic, caused by the two-way transfer of the state required for the intermediate buffering into the LANai internal memory, which could theoretically increase the latency of cache misses due to the increased traffic on the host data bus. However, as we will show in the experimental study in Chapter 5, semi-asynchronous checkpointing does not produce performance degradation even for simulation software exhibiting unusually low locality of references, case in which the frequency of cache misses is expected to be high.

The semi-asynchronous checkpointing functionalities of CCL are accessed by the following API:

**Initialization Functions**

- **void ckpt_init(int num_LPs, int state_size).** Upon the startup of the simulation, the mid-level sub-layer of SL must map reserved physical memory blocks into virtual address space, so that each LP can access its current state buffer through virtual addressing. We have implemented the mapping in the way that blocks of physical memory pages are seen as grouped into a single block of contiguous virtual memory pages. Then, the upper sub-layer partitions the reserved memory in order to assign a specific portion of the memory to each LP hosted by that machine. As already outlined, the portion associated with any LP is used for storing both the current state buffer of the LP and the stack of checkpointed states. Both mapping and partitioning are activated by calling the `ckpt_init()` function. Actually, this function performs the previously described mapping using the LINUX/UNIX system call `mmap()` on the device file `/dev/mem`. This shows that also this function is dependent on the host OS. Porting the software onto different OSs requires substitution of the `mmap()` system call function with an equivalent solution to map physical memory into virtual address space.

The execution of this function has also the effect to initialize data structures that, for each LP hosted by the machine, keep track of the busy entries in the stack of checkpointed states. Since the physical memory destined to the stack is actually managed as a circular buffer, keeping track of the busy entries means, in practice, keeping track of two indexes.

- **void *get_state_pointer(int LP_id).** Once mapping and partitioning have been executed, the LP can retrieve the virtual address associated with its current state buffer through the function `void *get_state_pointer(int LP_id)`, where LP_id is the identifier of the
LP \(^2\). Since the mapping between this virtual address and the corresponding physical address has already been managed by `ckpt_init()`, the programmer will be able to transparently access and modify any byte inside the reserved buffer.

Checkpointing Management Functions

- **int semi_async_ckpt(int LP_id, time_type simulation_clock).** From the point of view of the simulation application, issuing a semi-asynchronous checkpoint request means communicating to the LANai processor that the EBUS DMA must be programmed for the data transfer associated with checkpointing. This takes place through the API function `semi_async_ckpt(int LP_id, time_type simulation_clock)`, where LP\(_\text{id}\) is the identifier of the LP whose state needs to be checkpointed, and `simulation_clock` is the value of the current simulation time of the LP \(^3\). Since the library manages the checkpoint stacks of the LPs in a totally transparent way to the application programmer, LP\(_\text{id}\) is a sufficient parameter to identify both the state buffer and the entry into the stack of checkpoints that must be involved in the data transfer.

In addition, the execution of the `semi_async_ckpt()` function has the effect to communicate to the ECP the physical addresses of the state buffer of the LP and of the entry into the checkpoint stack (transparently mapped by the mid-level sub-layer). This is achieved by writing this information into a proper buffer located in the LANai internal memory.

- **int recover_state(int LP_id, time_type recovery_time).** Transparent management of checkpoint stacks during a rollback phase is supported by the API function `recover_state(int LP_id, time_type recovery_time)`, which reloads into the state buffer associated with LP\(_\text{id}\) the earliest checkpoint with simulation time less than or equal to `recovery_time`. Invocation of this function has also the effect to transparently prune the stack by all the checkpoints, if any, with simulation time larger than `recovery_time`.

- **int prune_stack(int LP_id, time_type global_virtual_time).** This function allows transparent storage recovery of busy entries of the checkpoint stack associated with LP\(_\text{id}\) during a fossil collection phase. Execution of this function has the effect to prune the checkpoint stack

\(^2\)LP identifiers must range between 0 and \(n - 1\), where \(n\) is the number of LPs hosted by the machine. Mapping into this range is required if a different identification is used at the simulation application level.

\(^3\)In the current version of CCL `time_type` is a redefinition of `double`. 
associated with LP\textunderscore id by all the checkpoints, except the earliest one, whose simulation time is less than or equal to global\textunderscore virtual\textunderscore time.

The re-synchronization functions are still left to describe. CCL offers different re-synchronization functions, each of which offering a distinct re-synchronization semantic. Specifically, the $C\mathcal{A}$ and $\mathcal{M}C$ semantics are implemented. There is no need for separate implementations of $AF$ and $\mathcal{C}A$ semantics, since, as already observed in Section 3.4.3 of Chapter 3, they are particular cases of $C\mathcal{A}$ obtained by imposing a threshold of 0 and 1, respectively. However, to fully explain such implementations, more details on the ECP are needed and will be presented in the following section. We will come back to the description of the re-synchronization functions in Sections 4.2.5 and 4.2.6.

4.2.4 Structure of the Extended Control Program

The control program run by the LANai processor has the responsibility to activate and control the three DMA engines on board of the card. To effectively support semi-asynchronous checkpointing this program must be structured in a way to ensure the Non-Intrusiveness requirement identified in Section 3.2 of Chapter 3. In this context, Non-Intrusiveness means that communication functionalities must not suffer from the activation of checkpointing functionalities. To achieve this, we have modified the control program’s conceptual structure described in Section 4.1.2. The new structure, namely the structure of the ECP, is shown in Figure 4.7.

Any checkpointing operation is split by the ECP into a sequence of invocations of the function \texttt{ckpt\textunderscore burst().} Each invocation has the effect to program the EBUS DMA to transfer up to a maximum amount of bytes, called \textit{burst}, from the LP state buffer ($sb_j$) to the LANai internal memory (intermediate buffering) or from the LANai internal memory to the checkpoint stack of the LP ($stack_j$).

Actually, splitting the checkpointing operation into a sequence of bursts avoids keeping the hardware on board of the myrinet card (i.e. the EBUS DMA, the PCI bridge and the LBUS) busy due to checkpointing for excessively long periods. As respect to this point, access to the EBUS DMA, the PCI bridge and the LBUS is required for block-DMA operations which transfer messages incoming from the network into the receive queue. Also, access to the PCI bridge and the LBUS is required for zero-copy sends issued by the application.

In addition, data transfer operations associated with checkpointing functionalities are activated only in case no block-DMA operation is currently required to transfer messages into the receive queue located onto host memory.
(see line 5 of the ECP), thus achieving lower priority of data transfer associated with checkpointing as compared to block-DMA. No preemption on data transfer associated with checkpointing could be used to favor communication since, according to hardware specification [60], we might incur problems with the PCI protocol.

The combination of both previous features (bursting of checkpointing operations and lower priority to checkpointing data transfers as compared to block-DMA) actually permits to respect the Non-Intrusiveness requirement. As respect to this point, in Section 4.3 we will show how to identify the maximum value for the burst length which allows the performance of communication functionalities not to be significantly perturbed by the activation of checkpointing functionalities.

As a last point related to Non-Intrusiveness, handling a single semi-asynchronous checkpoint request at a time is mandatory in order to ensure such a requirement. Specifically, in case multiple checkpoint requests were accepted, the ECP should be able either to handle time-sharing of these requests, or at least to queue them. Anyway, non-minimal time requirements for handling time-sharing or queuing might reduce responsiveness of the ECP as respect to the activation of DMA functionalities for supporting message transfer, with obvious negative impact on communication performance.

Finally, we note that by substituting lines 5 and 6 in Figure 4.7 with the statement <activate block-DMA>, conditioned to the predicate “block-DMA needed”, we obtain in practice a classical control program for fast speed messaging layers for myrinet.

### 4.2.5 Implementation of the CCA Re-synchronization Semantic

As already mentioned, any implementation of re-synchronization based on CCA relies on the ability to track the advancement status of an ongoing semi-asynchronous checkpointing operation. In the CCL software architecture, the only component able to track the advancement status of a checkpointing operation is the ECP run by the LANai processor. Such an ability derives from the fact that this program has the responsibility to handle (i.e. to activate and control) the EBUS DMA for data transfer operations associated with check-

```plaintext
1. While (1)
2. if (message needs to be sent) <activate send DMA>;
3. if (message needs to be received) <activate receive DMA>;
4. if (EBUS DMA not busy)
5. if (block-DMA not needed AND checkpoint burst needed) ckpt.burst();
6. if (block-DMA needed AND checkpoint burst not active) <activate block-DMA>;
```

Figure 4.7: High Level Structure of the ECP.
pointing. Given that, due to the Non-Intrusiveness requirement, any checkpointing operation issued at the application level is actually split by ECP into a sequence of EBUS DMA data transfer operations (each of which is responsible for the transfer of up to a maximum amount of bytes of the state, namely a burst, into the stack of checkpoints or into the LANai internal memory), a straightforward way to track the advancement of any ongoing checkpointing operation consists in counting the number of already completed EBUS DMA data transfers (from/to host memory) associated with that operation. To implement this solution, a counter has been introduced in the LANai internal memory, namely completed_transfers, which is managed as follows. The counter is reset by the function semi_async.ckpt() upon issuing a checkpoint request at the application level. It is incremented by the ECP each time the program becomes aware that an EBUS DMA data transfer (from/to host memory) associated with the checkpointing operation has been completed.

Efficiency of the implementation of this type of checkpoint advancement tracking mechanism derives from the fact that the overhead for handling the counter update is negligible in practice since the counter increment instruction is not time consuming. On the other hand, we note that the granularity (i.e. the effectiveness) of this tracking mechanism is obviously dependent on the selected value for the burst length. Issues related to this dependency will be discussed in Section 4.2.5.1.

The function int ckpt_cond_abort(float threshold) has been introduced in the CCL API to support the CCA re-synchronization semantic. The parameter threshold indicates the completion percentage of the last activated checkpointing operation, if any, under which the checkpointing operation must be aborted according to the CCA semantic. To decide whether to abort the checkpointing operation or not, ckpt_cond_abort() needs information about both the current value of the completed_transfers counter and the total number of EBUS DMA data transfers required for the operation. This information is maintained into an additional variable located onto host memory, namely total_transfers, visible to the function ckpt_cond_abort(), as well as to the function semi_async.ckpt(). In other words, this information is handled in a totally transparent way to the application programmer.

As soon as the checkpointing operation is issued by the application, the function semi_async.ckpt() computes the total number of EBUS DMA transfers (from/to host memory) to complete the operation according to the following expression:

\[
\text{total transfers} = 2 \times \left\lceil \frac{\text{state size}}{\text{burst length}} \right\rceil \tag{4.1}
\]

where the multiplier factor 2 takes into account the fact that data transfer associated with checkpointing needs intermediate buffering. The variable
total_transfers stores the obtained result, making it available to the function ckpt_cond_abort(). The function ckpt_cond_abort() takes the decision on whether to abort or not the last activated checkpointing operation on the basis of the condition \( \frac{\text{completed_transfers}}{\text{total_transfers}} < \text{threshold} \). The operation is aborted if the condition is verified.

Checkpoint abort requires notification of the abort decision to the ECP run by the LANai processor. This issue has been tackled by the use of a flag, namely ckpt_abort, implemented as a word located into the LANai internal memory. (A word has been used, instead of a single byte, for the flag ckpt_abort since the LANai 4 chip internal circuitry is optimized for aligned words access. Also, the host CPU accesses an aligned word using a single bus cycle, as it would be for the case of a single byte.) This flag is managed as follows. The flag is initialized to zero. It is set to one by the function ckpt_cond_abort() in case the checkpoint abort decision is taken. It is eventually reset by the ECP after it has interrupted EBUS DMA data transfer associated with checkpointing.

Abort of an ongoing checkpointing operation cannot take place as a pre-emption on an in-progress EBUS DMA data transfer since, as already pointed out, according to hardware specifications [60], it may cause problems to the PCI protocol. Therefore, abort is implemented by simply avoiding the activation of additional EBUS DMA transfers related to that checkpointing operation. Specifically, the ECP activates a new EBUS DMA transfer only in case the ckpt_abort flag has not been set to one by the function ckpt_cond_abort().

As a last point, if the abort condition (i.e. \( \frac{\text{completed_transfers}}{\text{total_transfers}} < \text{threshold} \)) is not verified, the function ckpt_cond_abort() behaves as follows. It returns immediately in case the checkpointing operation has already been completed. Otherwise it spinlocks around a flag ckpt_completed, also implemented as a word located in the LANai internal memory, managed as follows. It is set to zero by the function semi_asynch_ckpt(), upon issuing a checkpoint request (this flag is also used to indicate to the ECP that the information to handle the request has been already written into a proper buffer in the LANai internal memory). It is eventually set to one by the ECP run by the LANai processor as soon as the data transfer associated with the checkpoint request has been completed.

4.2.5.1 On the Effectiveness of the Checkpoint Advancement Tracking Mechanism

Since the granularity of the checkpoint advancement tracking mechanism previously presented is equal to a whole EBUS DMA data transfer (from/to the host memory), the Minimum Trackable Advancement Percentage (MTAP) of
a checkpointing operation depends on the total number of EBUS DMA data transfers required to complete the operation itself. More precisely, MTAP can be computed as

$$\text{MTAP} = \frac{1}{\text{total transfers}}$$  \hspace{1cm} (4.2)

Plugging (4.1) in (4.2), we get

$$\text{MTAP} = \frac{1}{2 \times \left\lceil \frac{\text{state size}}{\text{burst length}} \right\rceil}$$  \hspace{1cm} (4.3)

Expression (4.3) shows the dependency of MTAP on the size of the LP state to be checkpointed and on the burst length. To increase the precision of the tracking mechanism one could theoretically reduce the burst length. However, this is not a viable solution since, as it will be observed in the next section, in order to complete a checkpointing operation within reasonable latency, the burst length should be kept at the maximum value that still allows communication functionalities not to be significantly perturbed by the activation of checkpointing functionalities.

To provide the reader with insights on the effects of such a dependency for realistic burst lengths, we show in Figure 4.8 three different plots for MTAP obtained for burst length set to 512 bytes, 1 Kbyte and 2 Kbytes \footnote{The study reported in Section 4.3 will show that, for the specific cluster environment used in the experimental part of this dissertation, the maximum burst length which does not cause significant perturbation of the performance of communication functionalities is expected to be about 1 Kbyte}, while varying LP state size between 1 Kbyte and 10 Kbytes (with step of 1 Kbyte).

The plots show that MTAP is in the order of 0.25 or less except for the case of small state size (i.e. about 1 or 2 Kbytes). However, the reduced precision of the tracking mechanism for the case of small size states should not be a problem in practice. Specifically, for small size states, the likelihood for the semi-asynchronous checkpointing operation to be not yet completed when the re-synchronization functionality is activated is expected to be minimal. This is confirmed by results we report in Section 5.1 of Chapter 5. Therefore, for small state size the CCA semantic is actually insensitive to the precision of the tracking mechanism, thus allowing the mechanism not to exhibit real ineffectiveness.

### 4.2.6 Implementation of the $\mathcal{MC}$ Re-synchronization Semantic

As already observed in Section 3.4.4 of Chapter 3, solving the checkpointing-recovery cost model in expression (3.3) underlying the $\mathcal{MC}$ re-synchronization semantic, requires knowledge of both classical and non-classical parameter
values. In the same section we have already seen how classical parameters may be determined. However, any implementation of \( \mathcal{M} \) requires also information about the value of what we defined as the non-classical parameters (namely \( \Delta_{\text{complete}} \) and \( \Delta_{\text{interrupt}} \)), dependent on the specific implementation of semi-asynchronous checkpointing.

We shall now proceed showing how to cope with the values of non-classical parameters in the specific CCL implementation. Subsequently, details on the implementation of the function supporting \( \mathcal{M} \) are reported.

Two non-classical parameters appear in the cost model in (3.3), namely the expected residual completion latency of the checkpointing operation \( \Delta_{\text{complete}} \) and the time to notify the abort decision to the ECP, namely \( \Delta_{\text{interrupt}} \). We discuss in this section how to deal with these parameters in CCL.

**Determining \( \Delta_{\text{interrupt}} \).** As already discussed in Section 4.2.5 while presenting the implementation of CCA semantic, checkpoint abort only requires notification of the abort decision to the ECP. This has been implemented by letting the function `ckpt_cond_abort()` (run at the host side) set a flag, namely `ckpt_abort`, implemented as a word located into the LAI internal memory. Flag setting indicates to the ECP that any ongoing checkpointing operation must be interrupted. Actually, software run at the host side must execute no other action for handling checkpoint abort since all the work for aborting the checkpointing operation (i.e. for interrupting data transfer asso-
associated with checkpointing) is carried out by the ECP.

We maintain this approach also for the implementation of the $\mathcal{MC}$ semantic, therefore the only overhead experienced at the host side for handling checkpoint abort is the latency to set the flag value into the LANai internal memory. $\Delta^{\text{interrupt}}_{\text{ckpt}}$ corresponds exactly to this latency value, evaluated in case the last activated semi-asynchronous checkpointing operation has not yet been completed. As respect to the last statement, we recall that, by the definition of $\mathcal{MC}$ in Figure 3.5 of Chapter 3, the cost model must be solved only in case the last activated semi-asynchronous checkpointing operation is still ongoing, therefore we need the value of $\Delta^{\text{interrupt}}_{\text{ckpt}}$ just in case the EBUS DMA is active either for data transfer associated with checkpointing or for higher priority transfers of messages incoming from the network into the receive queue.

The host CPU and the EBUS DMA have the same access priority to the LBUS. Also, other components on board of the card (e.g. the LANai processor and the other DMA engines) have lower access priority to the LBUS [60]. As a consequence, given that the flag is set using a single bus cycle, we expect minimal variance for the latency to set the flag value despite the fact that the EBUS DMA is working either for data transfer associated with checkpointing or for message transfer into the receive queue. Specifically data transfer currently performed by the EBUS DMA can delay the flag update performed by the host CPU of at most a single bus cycle.

As an empirical support to previous deduction, we have measured the time to set the flag value in case the EBUS DMA continuously performs data transfer operations from/to the host memory involving data blocks of different sizes ranging from 4 bytes to 8 Kbytes. The sequence of activations of the EBUS DMA is such that a data block is first copied from the host memory into the LANai internal memory and then is copied back from the LANai internal memory into the host memory. The measures have been taken for a M2M-PC132 myrinet network card mounted on a Pentium II 300 MHz running LINUX (kernel version 2.0.32), which is the same architecture we will use for the experimental study in Chapter 5. The results reported in Table 4.1 show that the latency to set the flag value is independent of the data block size being transferred by the EBUS DMA. Therefore, such a value can be used as a reliable measure for $\Delta^{\text{interrupt}}_{\text{ckpt}}$ while solving the checkpointing-recovery cost model.

<table>
<thead>
<tr>
<th>Data Block Size (Bytes)</th>
<th>4</th>
<th>16</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag Setting Latency (Microseconds)</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.83</td>
<td>0.84</td>
<td>0.84</td>
</tr>
</tbody>
</table>

Table 4.1: Flag Setting Latency vs Data Block Size (Average Over 1000 Samples).
From a pragmatical point of view, this latency value can be measured once upon the installation of CCL on the specific hardware/software architecture. The measurement can be performed through adequate benchmarks, such as the one we have used for the previously described experiments. Then the function implementing the \( \mathcal{MC} \) re-synchronization semantic can use that measured value to solve the cost model. Determining the value of \( \Delta_{\text{interrupt}}^{\text{ckpt}} \) off-line upon the installation of CCL avoids any additional overhead for the treatment of this parameter value while the simulation application is in progress.

**Determining** \( \Delta_{\text{ckpt}}^{\text{complete}} \). By the description in Section 4.2.4, we know that any semi-asynchronous checkpointing operation is split by the ECP into a sequence of data transfer operations (bursts) from the host memory, i.e. from the LP state buffer, to the LANai internal memory and vice-versa, i.e. from the LANai internal memory to the stack of checkpoints located in the host memory. The counters \textit{total\_transfers} and \textit{completed\_transfers} maintained by CCL to implement \( \mathcal{CCA} \) (already described in Section 4.2.5) indicate the total amount of bursts required by the last activated semi-asynchronous checkpointing operation and the amount of those bursts which have already been completed. As a consequence, the difference between the two counter values indicates the amount of bursts that still need to be carried out in order to complete the checkpointing operation.

Denoting with \( \Delta_{\text{burst}} \) the expected time for a single burst performed by the EBUS DMA, then we get that the lower bound for \( \Delta_{\text{ckpt}}^{\text{complete}} \), namely \( L \), is

\[
L = (\text{total \_transfers} - \text{completed \_transfers}) \Delta_{\text{burst}} \quad (4.4)
\]

This lower bound is obtained in case no message needs to be transferred into the receive queue through the EBUS DMA during a period that starts just upon re-synchronization occurrence and ends with the completion of the checkpointing operation, which we will refer to as the \textit{re-synchronization period}. However, it is possible that during that period some messages incoming from the network need to be transferred into the receive queue through the EBUS DMA. Since message transfer has higher priority as compared to data transfer associated with checkpointing, the real value of \( \Delta_{\text{ckpt}}^{\text{complete}} \) might be actually larger than the lower bound \( L \). The real value of \( \Delta_{\text{ckpt}}^{\text{complete}} \) is actually equal to the length of the previously mentioned re-synchronization period. Our objective is therefore to evaluate the expected length of the re-synchronization period.

Denoting with \( M \) the number of messages already buffered into the LANai internal memory upon re-synchronization occurrence and waiting to be transferred into the receive queue through the EBUS DMA, and with \( \Delta_{\text{message}} \) the expected time required by the EBUS DMA for transferring a single message.
from the LANai internal memory into the receive queue, we get that the length of the re-synchronization period, namely \( \Delta_{\text{checkpoint}}^{\text{complete}} \), can be expressed as

\[
\Delta_{\text{checkpoint}}^{\text{complete}} = L + M \Delta_{\text{message}} + L'
\]

(4.5)

where \( L \) is the previously mentioned lower bound, \( M \Delta_{\text{message}} \) is the expected time for transferring those \( M \) messages into the receive queue through the EBUS DMA, and \( L' \) is the additional time required for transferring into the receive queue the additional messages incoming from the network during the re-synchronization period.

Denoting with \( f \) the expected frequency of message arrival from the network during the re-synchronization period we get

\[
L' = f \Delta_{\text{checkpoint}}^{\text{complete}} \Delta_{\text{message}}
\]

(4.6)

where \( f \Delta_{\text{checkpoint}}^{\text{complete}} \) represents the amount of messages incoming from the network during the re-synchronization period.

Plugging (4.4) and (4.6) in (4.5), and explicitating \( \Delta_{\text{checkpoint}}^{\text{complete}} \) we obtain the following expression

\[
\Delta_{\text{checkpoint}}^{\text{complete}} = \frac{(\text{total transfers} - \text{completed transfers}) \Delta_{\text{burst}} + M \Delta_{\text{message}}}{1 - f \Delta_{\text{message}}}
\]

(4.7)

By (4.7), calculation of \( \Delta_{\text{checkpoint}}^{\text{complete}} \) requires knowledge of \( M, \Delta_{\text{burst}}, \Delta_{\text{message}} \) and \( f \). The implementation already maintains a counter in the LANai internal memory (managed by the ECP and accessible by software run at the host side) which keeps track at any instant of the amount of messages already buffered into the LANai internal memory, which need to be transferred into the receive queue. (The counter value is used by the ECP to manage the activation of block-DMA operations for transferring those messages.) Therefore, the value of \( M \) is straightforwardly available to the re-synchronization function implementing \( \mathcal{MC} \) just through this counter value.

In case a commit decision is taken for an ongoing checkpoint operation, the re-synchronization function must wait for the completion of the operation in a way to temporarily suspend any other simulation operation carried out by the CPU. As for the case of \( \mathcal{CCA} \) described in Section 4.2.5, the re-synchronization function we have developed to implement \( \mathcal{MC} \), to be described later, spinlocks around the flag `ckpt_complete`, which is set to one by the ECP as soon as the operation is completed. As a consequence, the values of \( \Delta_{\text{burst}} \) and \( \Delta_{\text{message}} \) must be evaluated considering the interference on EBUS DMA activities due to access of the host CPU to the PCI bridge and to the LBUS caused by read operations continuously issued on the flag while waiting for the
completion. (All the other components on board of the card have lower access
priority to the LBUS as compared to the EBUS DMA [60], therefore they do
not produce interference on data transfer performed by the EBUS DMA).

As respect to this point, measures reported in Section 4.3.2.1 have shown
that the time to transfer a data block from/to the host memory through the
EBUS DMA while the host CPU spinlocks around a flag located in the LANai
internal memory increases linearly with the data block size. As a consequence,
one fixed the message size and the burst size, $\Delta_{burs}$ and $\Delta_{mess}$ can
be assumed as constant values any time we hypothesize that no other PCI pe-
ripherals (such as audio cards) are using the same PCI bus. Since this can be
considered as a common approach when dedicating a cluster to a specific par-
allel computing application, $\Delta_{burs}$ and $\Delta_{mess}$, as in the case of $\Delta^{\text{interrupt}}_{ckpt}$,
can be assumed as constant values, to be determined while installing CCL and
made available to the re-synchronization function implementing $\mathcal{M}_C$.

By previous arguments, $f$ is the only parameter that depends on proper
dynamics of the specific parallel execution, whose value really needs to be
determined upon re-synchronization occurrence. As pointed out before, $f$
is the expected frequency of message arrival during the re-synchronization
period, however we need this value at the start of the period itself in order to
solve the cost model, therefore a prediction mechanism for determining this
value must be encompassed.

According to the commonly accepted belief that the recent past behavior
should be a reliable indicator of the immediate future behavior (for instance
this is also the basic assumption underlying many OS dispatchers design, where
recent process/thread behavior, in terms of CPU bursts or I/O requests is as-
sumed as representative of the immediate future behavior [82, 88]), we have
decided to approximate $f$ with the frequency of message arrival from the
network in the interval between the activation of the last semi-asynchronous
checkpointing operation and the occurrence of re-synchronization. In other
words, the prediction of the value of $f$ is based on statistics related to the
temporal window associated with the already executed portion of the check-
pointing operation involved in the re-synchronization occurrence.

To compute that frequency value, we further exploit hardware features
of the LANai 4 chip. Specifically, this chip is equipped with a Real Time
Clock register (RTC), providing real time measures with granularity on the
order of 0.5 microseconds, which can be reset/read by the host CPU. We
have introduced an additional counter in the LANai internal memory, namely
incoming messages, which counts the amount of messages incoming from
the network since the activation of the last semi-asynchronous checkpointing
operation. The counter is incremented by the ECP each time it activates
the receive-DMA on board of the card. The function semi_async_ckpt()
resets both incoming messages and the register RTC upon the issue of a
semi-asynchronous checkpointing operation. The value of $f$ to be used while computing $\Delta_{\text{ckpt}}^{\text{complete}}$ can be evaluated by the re-synchronization function implementing $\mathcal{MC}$ as

$$ f = \frac{\text{incoming messages}}{\$RTC} $$

(4.8)

As a last observation, we note that this estimation method for $f$ is expected to produce in practice negligible additional overhead due to the fact that the time for resetting the values of RTC and incoming messages is a negligible percentage of the whole execution time of the function $\text{semi\_async\_ckpt}()$, despite reset itself needs access to the LANai internal memory passing through the PCI bridge. Specifically, upon its execution, the function $\text{semi\_async\_ckpt}()$ must provide to the ECP run by the LANai processor the information required for programming the EBUS DMA for data transfer operations associated with checkpointing. This already happens through a sequence of accesses to the LANai internal memory in order to write that information into a proper buffer. Also, the overhead on the ECP for incrementing the counter incoming messages is actually negligible in practice being it implemented through few machine instructions.

The $\mathcal{MC}$ Re-synchronization Function

The re-synchronization function implementing $\mathcal{MC}$ we have embedded in CCL has the prototype $\text{int min\_cost(double prob, double cumulate, int option)}$, where $\text{prob}$ corresponds to the value of the probability $P(X)$ in the cost model in expression (3.3), $\text{cumulate}$ is the sum of the execution costs of all the events executed by the LP since the last committed checkpointing operation expressed in microseconds (also this quantity appears in the cost model), and $\text{option}$ is a value we can pass to the function, which determines, as we shall discuss below, the run-time behavior of the $\text{min\_cost()}$ function.

In practice we have left all the classical parameters of the cost model (i.e. those mentioned in Section 3.4.4 of Chapter 3) as parameters to be passed to $\text{min\_cost()}$. Instead, all the non-classical parameters (previously treated in this subsection) are handled by CCL in a transparent way to the application programmer, according to the solutions previously described. This is an engineering choice allowing the application level programmer to use the preferred solution, for example one among those listed in Section 3.4.4 of Chapter 3, to tackle the determination problem of the classical parameters.

As already pointed out in Section 3.4.4 of Chapter 3, the value of the parameter $\Delta_{\text{reload}}$ is not required for solving the cost model as the difference between $x$ and $y$ (see line 3 of the algorithm in Figure 3.5, defining the $\mathcal{MC}$
semantic). This is the reason why we have introduced no parameter for passing the value of $\Delta_{\text{reload}}$ to the \texttt{min\_cost()} re-synchronization function.

As respect to \texttt{option}, its value can be selected among macros in the set \{\texttt{DEFAULT, ALWAYS\_FREEZE, ALWAYS\_ABORT}\}, and determines one of the following three run time behaviors:

\textbf{DEFAULT} - the \texttt{min\_cost()} function commits/aborts the checkpoint on the basis of the $\mathcal{MC}$ semantic, exactly as defined by the algorithm in Figure 3.5.

\textbf{ALWAYS\_FREEZE} - the \texttt{min\_cost()} function always waits for the completion of any ongoing checkpointing operation. This is done by simply spinlocking around the flag \texttt{ckpt\_completed}. In practice, we get an emulation of the Application Freezing (AF) re-synchronization semantic discussed in Chapter 3.

\textbf{ALWAYS\_ABORT} - the \texttt{min\_cost()} function always aborts an ongoing checkpointing operation. This is done by simply setting the flag \texttt{ckpt\_abort} in the LANai internal memory. This allows getting an emulation of the Checkpoint Abort (CA) re-synchronization semantic also discussed in Chapter 3.

The DEFAULT value for \texttt{option} is straightforward and does not deserve further discussion. The reason why we have introduced the other two options is instead discussed in the following.

\textbf{ALWAYS\_FREEZE option.} As introduced in Section 1.3.2 of Chapter 1, the earliest simulation time for a rollback occurrence in an optimistic parallel simulation is GVT. Therefore, in order to correctly support state recovery, each LP must retain the latest checkpointed state value associated with simulation time $T$ less than or equal to GVT and also all the events with timestamp larger than $T$ (these events must be available in case coasting forward must be executed starting from simulation time $T$). Therefore, that checkpoint and all those events cannot be discarded during the execution of the fossil collection procedure. As discussed in Section 2.1.4 of Chapter 2, if very few states are recorded in the course of the simulation, then it is possible that a large amount of events must be retained. The drawback incurred is that the amount of memory recovered during any fossil collection may be small, thus there is the risk that the GVT calculation and the fossil collection procedure must be executed frequently (as memory saturates frequently). This may have detrimental effects on performance.

If the commitment of checkpoints is determined only on the basis of the $\mathcal{MC}$ semantic (i.e. the function \texttt{min\_cost()} is always executed with the DE-
FAULT value for option), then there is the possibility that very few checkpoints are committed for a given LP. (This might happen, for example, when the parameter \( P(X) \) in the cost model approaches the value zero, and upon re-synchronization occurrence, we frequently find that the last activated checkpointing operation is still ongoing.) In this case we might get a negative interference with the frequency of GVT calculation and fossil collection. In order to prevent this problem, we can force commitment of an ongoing checkpointing operation for an LP in case the number of executed events from the last committed checkpoint of that LP oversteps a given threshold (see [31, 71, 80] for the determination of such a threshold). This can be done just exploiting the \texttt{ALWAYS\_FREEZE} option for \texttt{min\_cost}().

\textbf{ALWAYS\_ABORT option.} The re-synchronization function must be called to avoid data inconsistency whenever the LP associated with the last activated checkpointing operation is re-scheduled for event execution. In case the scheduled LP must roll back, and the checkpointing operation is still ongoing upon re-synchronization, it is better to abort the operation. This is because the current value of the LP state (which is being checkpointed) represents incorrect information to be discarded (i.e. overwritten) during the state recovery procedure. As a consequence committing that checkpointing operation would only result in a useless checkpoint to be eventually discarded by the checkpoint stack. To avoid committing that checkpointing operation, the \texttt{ALWAYS\_ABORT} option can be used in this kind of situation.

As a last observation, we note that the function \texttt{ckpt\_cond\_abort()} does not need the option value since always freezing/abort behaviors can be obtained by adopting adequate values of the parameter \texttt{threshold}, namely 0/1.

### 4.3 Low Level Parameter Tuning

Splitting the checkpointing operations in bursts and assigning lower priority to EBUS DMA data transfers associated with checkpointing, as compared to block-DMA, is a basic design choice for CCL, which is expected to allow checkpointing not to significantly interfere with the performance of communication functionalities offered by CCL. In other words, this is the design choice adopted to guarantee the Non-Intrusiveness requirement identified in Section 3.2 of Chapter 3. However, a well suited burst length should be selected in order for this to really happen. In this section we discuss how to determine such a well-suited burst length.

We first provide a detailed identification of performance issues associated with the burst length. Then a quantification of effects related to these issues, obtained through proper benchmarking, allowing a well suited burst selection
is reported for the specific cluster environment adopted in the experimental study reported in Chapter 5. The identification methodology and the used benchmarking can be anyhow re-used for determining the burst length to be adopted on other architectures.

4.3.1 Performance Issues Identification

4.3.1.1 Burst Length vs Block-DMA Activation Delay

The EBUS DMA is used by CCL to support two distinct operations in different time intervals. Specifically, it is used for block-DMA transfer of incoming messages into the receive queue (see line 6 of the ECP) and also for data transfer operations (from/to the host memory) associated with checkpointing (see line 5 of the ECP).

As pointed out in Section 4.2.4, the latter operation has lower priority as compared to block-DMA data transfer, however no preemption is exercised. This means that messages incoming from the network must be maintained into the LANai internal memory until any in-progress EBUS DMA based data transfer operation (from/to the host memory) associated with checkpointing is not yet completed. The burst length determines the amount of bytes to be managed by the transfer operation, therefore it might delay the activation of block-DMA operations to transfer messages into the receive queue. Very long bursts might cause intolerable delay in the message delivery, which, in turn, might affect the overlaying applications. As respect to the simulation application, [12] has shown how delaying message delivery might have strong negative impact on performance due to a possible increase in the amount of rollback.

4.3.1.2 Burst Length vs Zero-Copy Send Latency

EBUS DMA based data transfer operations associated with checkpointing might interfere also with send operations. Specifically, send operations are based on the zero-copy approach, that requires access to the PCI bridge to copy the message content into the LANai internal memory. Very long bursts might negatively interfere with the latency for copying the message into the LANai internal memory since the copy of each word of the message might experience additional costs due to contention on the PCI bridge. As a consequence, there is the risk that: (i) the overlaying applications suffer from an increase in the send operation overhead, and (ii) the message transfer delay increases due to the increased latency of copying the message content into the LANai memory.
4.3.1.3 Burst Length vs Checkpointing Latency

Effects of the burst length on the completion time of the checkpointing operation are related to the scheduling sequence of block-DMA operations and EBUS DMA data transfer operations associated with checkpointing. The scheduling sequence is determined by the ECP run by the LANai processor in the way that any EBUS DMA data transfer operation associated with checkpointing has lower priority as compared to block-DMA.

The activation of each EBUS DMA based data transfer operation associated with checkpointing experiences a given scheduling delay determined by the decisions taken by the ECP, which favors block-DMA operations. Therefore, completion of a single checkpointing operation through many EBUS DMA based data transfer activations determines a non-minimal checkpointing latency due to the fact that each activation adds its scheduling delay to the completion time of the checkpointing operation. To mitigate the effects of scheduling delay, the use of very long bursts is recommended.

Overall, the longer the burst, the lower the expected completion time of the checkpointing operation. Therefore, with very long bursts we get a decrease of the probability that the last checkpointing operation is already completed upon the invocation of the re-synchronization function.

4.3.1.4 Issues Summary

By the arguments in previous sections, we summarize below well suited indications for the selection of the burst length, which are expected to alleviate each of the previously pointed out problems. By these indications we argue that, in general, a short burst should be selected, but not so short to incur unacceptable checkpointing latency.

<table>
<thead>
<tr>
<th>Block-DMA Activation Delay</th>
<th>Zero-Copy Send Latency</th>
<th>Checkpointing Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>Short</td>
<td>Long</td>
</tr>
</tbody>
</table>

4.3.2 Experimental Analysis

The second step of our methodology consists in putting the issues identified above to a practical use. Specifically, the identification above determined a set of relevant phenomena that must be taken into account to determine a well value for the burst length. We now proceed to quantify these phenomena for a specific myrinet-connected cluster environment. Specifically each machine of the cluster is a Pentium II 300 MHz (128 Mbytes RAM - 512 Kbytes second level cache) running LINUX (kernel version 2.0.32) and equipped with M2M-PCI32C myrinet cards. This architecture is the same that we will use for the experimental study in Chapter 5.
Note that the analysis methodology we employ (e.g., software and system conditions) is general and can be used whatever is the real hardware/software architecture for which a well suited tuning of CCL must be performed.

4.3.2.1 Burst Length vs Block-DMA Activation Delay

To determine the effects of the burst length on the activation delay of block-DMA operations, we have measured the time needed to perform an EBUS DMA based data transfer operation while varying the size of the data block involved in the operation. Actually, the transfer latency for a given data block size is a measure of the worst case delay in the activation of block-DMA when the burst length is equal to that data block size. This is because the worst case occurs just when a block-DMA transfer is delayed for the whole time interval associated with an EBUS DMA based data transfer operation due to checkpointing.

The measures we report are related to the case of both transfer from the LANai internal memory to the host memory and vice-versa (recall that EBUS DMA based checkpointing requires data transfer in both directions).

![EBUS DMA Data Transfer Latency vs Data Block Size](image)

Figure 4.9: EBUS DMA Data Transfer Latency vs Data Block Size (Average of 1000 Samples).

Plots in Figure 4.9 show that the data transfer operation is completed within the same latency for both transfer directions (from/to host memory). Furthermore, they indicate that the data transfer time is bounded by about 2 microseconds for data blocks up to 128 bytes, and is bounded by 5 and 8 microseconds for data blocks up to 512 and 1024 bytes, respectively. Also,
the plots indicate that the transfer delay increases linearly vs the data block size (the logarithmic scale does not outline the linear behavior, but, at the same time, helps in providing plots for a very large interval of non-equidistant values of the data block size).

4.3.2.2 Burst Length vs Zero-Copy Send Latency

To evaluate the effects of the burst length on the latency of zero-copy sends issued by the application, we have used a control program run by the LANai processor which has been derived as a modification of the ECP reported in Section 4.4. It is structured as follows

1. While (1)
2. if (EBUS DMA not busy) ckpt_burst();

In other words, the only responsibility of the modified control program is to check whether the EBUS DMA is not busy and, in the positive case, to execute the function ckpt_burst() in order to activate the EBUS DMA for an operation that emulates a data transfer associated with checkpointing. The sequence of activations of the EBUS DMA is such that a block of bytes is first copied from the host memory into the LANai internal memory and then is copied back from the LANai internal memory into the host memory. In practice, the execution of this control program emulates a case in which data transfer operations associated with checkpointing are continuously executed. Therefore, it represents a “worst case scenario” in which access to the PCI bridge is continuously required for EBUS DMA based data transfer operations associated with checkpointing.

Fixed this scenario, we have measured the latency required for a zero-copy send operation. Specifically, we have measured the latency required by the host to copy the message content into the LANai internal memory and to set the data structures whose values indicate that a new message needs to be sent. We have measured that latency for the case of three different message sizes, namely 32, 64 and 128 bytes (5). The independent parameter in the analysis is the data block size transferred through a single EBUS DMA activation, namely the burst length characterizing the simulated data transfer operation associated with checkpointing.

The results are plotted in Figure 4.10 (dashed lines indicate reference latency values measured for the case of zero-copy sends executed with no active EBUS DMA based data transfer operation). They point out that, for data block size up to 1 Kbyte, the zero-copy send latency shows no relevant increase, as compared to the latency measured for the case of no active EBUS

---

5Investigation for message size up to 128 bytes is representative since optimistic PDES typically requires transfer of small size messages.
Figure 4.10: Zero-Copy Send Latency vs Data Block Size (Average of 1000 Samples).

DMA based data transfer, especially for the case of 32 and 64 bytes message size. On the other hand, data block size larger than 1 Kbyte might cause significant increase in the latency of the zero-copy send operation.

4.3.2.3 Burst Length vs Checkpointing Latency

To study the effects of the burst length on the latency of a checkpointing operation we have measured the latency for checkpointing a state of $X$ bytes, using burst length equal to $Y$. In the analysis we have varied $X$ from 256 bytes to 10 Kbytes and we have used different values for $Y$ ranging between 256 bytes and 5 Kbytes.

The plots in Figure 4.11 show that, as expected, the checkpointing latency decreases while the burst length increases. However, the checkpointing latency for any given state size $X$ does not decrease linearly vs the burst length $Y$. As an example, very strong latency reduction (up to 50%) is obtained changing $Y$ from 256 to 512 bytes. Instead, when $Y$ is changed from 512 bytes to 1 Kbyte, only a 25% additional reduction of the latency is noted. By this behavior we argue that, beyond a given threshold, long or very long burst lengths are likely to not originate very different checkpointing latencies. Therefore, to bound the checkpointing latency, the important thing is to avoid the use of minimal burst lengths (e.g. 256-512 bytes for the particular system settings).
Figure 4.11: Checkpointing Latency for Different Burst Lengths (Average of 1000 Samples).

### 4.3.3 Deductions

By the results in previous sections, burst length up to 512-1024 bytes produces almost negligible penalty in the zero-copy send latency (especially for message size up to 64 bytes), and, at the same time produces an increase in the worst case block-DMA activation delay that we expect to not negatively interfere in practice on the behavior of overlying applications. As respect to the latter point, when the burst is set to 1 Kbyte, the worst case delay for the activation of block-DMA increases from 3 to 8 microseconds. Although the increase is in the order of more than 200%, few additional microseconds in the delivery of messages to the overlying applications should not result in any perceptible effect, unless the application granularity, in terms of CPU time between message receipts, were itself in the order of few microseconds. On the other hand, using burst length of at least 1 Kbyte should not produce excessively long checkpointing latency. Therefore, we argue that 1 Kbyte is a well suited burst length to select at compile time of CCL in order to ensure adequate performance for both checkpointing and communication functionalities for the specific cluster environment considered in the analysis of Chapter 5.
Chapter 5

Experimental Results

“If you can't measure it, it's not science.”
Robert Heinlein, The door into summer

This chapter reports an experimental study of semi-asynchronous checkpointing, based on data collected using the implementation presented in Chapter 4, namely CCL. The study is structured in two separate sections:

1. In the first section we report the results obtained on a classical parameterized simulation benchmark in several different configurations, in order to demonstrate the potential of semi-asynchronous checkpointing in a variety of simulation settings.

2. In the second section a case study on a real world simulation application, namely a mobile system application, is reported.

All experiments presented in this chapter were performed on a cluster of Pentium II 300 MHz (128 Mbytes RAM - 512 Kbytes second level cache). All the PCs of the cluster run LINUX (kernel version 2.0.32) and are equipped with M2M-PCI32C myrinet cards. On the basis of the results presented in Section 4.3 of Chapter 4, burst length of 1 Kbyte has been selected for this cluster environment to support EBUS DMA data transfer associated with semi-asynchronous checkpointing.

The simulation software we have used implements the events as a compound structure with several fields (sender, receiver, timestamp etc.). For both the synthetic benchmark and the real world application, CCL delivers any message carrying an event to the recipient within about 20/25 microseconds when no congestion occurs on the switch. The same delay characterizes the transmission of antimessages. Message exchange among LPs hosted by the same machine does not involve operations of the CCL layer. There is an
instance of the optimistic simulation engine on each machine. Each LP is implemented as an application-level thread. The engine manages the local event list (resulting as the logical collection of the event lists of the local LPs) and schedules LPs for event execution according to the Smallest-Timestamp-First algorithm [47]. Memory space for new entries into the event lists of the LPs is allocated dynamically using classical \texttt{malloc()} calls. Therefore there is no pool of pre-allocated buffers for the event list management. Also, the event lists are implemented as simple linked lists. Antimessage sending is implemented by following the aggressive policy that is, antimessages are sent as soon as an LP rolls back [36]. Fossil collection is executed periodically.

5.1 Benchmark Study

In this section we report an experimental analysis of semi-asynchronous checkpointing conducted using the PHOLD synthetic benchmark. This study aims at observing the behavior of the semi-asynchronous mode while varying simulation application settings. The PHOLD benchmark, originally presented in [34], consists of a fixed number of LPs and of a constant number of messages (jobs) circulating among the LPs, which is referred to as message population. A message simply triggers the production of a new message with an increased timestamp value. Both the routing of messages among the LPs and the timestamp increments are taken from some stochastic distributions. Although a set of standard benchmarks for PDES does not exist, PHOLD is in practice one of the most used ones in the PDES literature [22, 24, 76, 80, 87, 98]. Before entering the analysis, we report a description of the testing methodology employed.

5.1.1 Testing Methodology

As we have seen in Section 3.5 of Chapter 3, \( \mathcal{CA} \) and \( \mathcal{MC} \) semantics for resynchronization determine a particular type of sparse checkpointing based on a-posteriori skipping of checkpoints. As respect to this point, we have decided to split the analysis with the PHOLD benchmark into two parts.

In the first one, namely Part A, we study the effects of a-posteriori skipping with no interference due to a combined use of a-priori checkpoint skipping. Specifically, the results for semi-asynchronous checkpointing reported in Part A have been obtained by imposing always positive result to the test in line 9 of the simulation engine in Figure 3.3, so that a semi-asynchronous checkpointing operation is requested for an LP after the execution of each simulation event (i.e. no a-priori checkpoint skipping is ever performed).

Both \( \mathcal{CA} \) and \( \mathcal{MC} \) semantics have been taken into consideration. In particular, to test the \( \mathcal{CA} \) semantic, in this part of the study we have ini-
tially treated threshold (i.e. the argument to the re-synchronization function \texttt{ckpt\_cond\_abort()}) as the independent parameter. Therefore, we have tested the behavior of semi-asynchronous checkpointing with \textit{CCA} assuming a set of different values for threshold, manually moving it from one value to another while performing the experiments. In Section 5.1.2.6 we have also addressed the issue of run-time adaptive tuning of the threshold parameter. On the other hand, the testing of the \textit{MC} semantic has been conducted by treating the cost model classical parameters as follows: (i) in the experiments the expected event granularity is known in advance, therefore we have used such an expected value as the execution cost \( \Delta_e \) of any event in the set \( EV(X) \) moving the LP state from the last checkpointed state value preceding \( X \) to the state value \( X \) (ii) the parameter \( P(X) \), namely the probability of rollback to a given state value \( X \), has been simply estimated as the rollback frequency of the LP in order to favor simplicity. Also, the solutions described in Section 4.2.6 of Chapter 4 have been adopted for non-classical parameters characterizing the specific implementation of semi-asynchronous checkpointing supported by CCL.

The second part of the study, namely Part B, is instead devoted to the observation of the effect of combining both a-posteriori and a-priori checkpoint skipping.

The following measures are reported:

- The event rate, that is the number of committed events per time unit, which is representative of the overall simulation execution speed.

- The frequencies \( F_1 \) and \( F_2 \) of checkpoint abort upon the invocation of the re-synchronization function. \( F_1 \) relates to re-synchronization in line 4 of the optimistic simulation engine in Figure 3.3, while \( F_2 \) relates to re-synchronization line 10. These frequencies have been evaluated as the ratio between the amount of checkpoint aborts at the corresponding re-synchronization point, and the total amount of issued semi-asynchronous checkpointing operations.

- The average distance (number of events) between two consecutive committed checkpoints of the same LP. This parameter allows us to measure the “density of committed checkpoints” (as respect to the amount of executed events) at the point where the event rate, namely the performance, is maximized.

Additional metrics will be introduced, whenever required, in some parts of the analysis.

We test semi-asynchronous checkpointing (both \textit{CCA} and \textit{MC} resynchronization semantics) against a classical Periodic State Saving (PSS) method.
based on CPU charged (synchronous) checkpointing, which has been selected as a representative sparse checkpointing methods relying on a-priori skipping of checkpoints. Although for some particular simulation problems (e.g. simulations with regular patterns in the difference between timestamps of consecutive events) performance improvements over PSS can be achieved by relaxing the constraint that checkpoints must be taken on a periodic basis [76], PSS remains, in general, a performance effective method. For this method, we report both the peak event rate and the corresponding value of the checkpoint interval \( \chi \). In our experiments, CPU charged checkpointing is implemented efficiently as a `memcpy()` call that copies the LP state into the stack of checkpointed states of that LP. As CCL adopts reserved main memory pages for both the LP’s states and their stacks, to ensure fairness in the comparison we have used the same approach for the case of CPU charged checkpointing.

Mean value analysis is employed, and each reported parameter value results as the average over 10 runs, all done with different random seeds. At least \( 2 \times 10^6 \) committed events were simulated in each run.

5.1.2 Part A

5.1.2.1 Basic Test Case

As a basic test case we have considered a PHOLD benchmark with 32 LPs evenly distributed on 4 machines of the previously described cluster. The message population has been fixed at 1 message per LP and messages are equally likely to be forwarded to any LP, with timestamp increments following an exponential distribution with mean 10 simulation time units.

In the experiments we have fixed the event execution time at about 150 microseconds, obtained by structuring the event routine as a simple CPU busy loop [71, 87], which has been selected as an intermediate event granularity value for parallel discrete event simulation applications. For simulation applications with very coarse event granularity, the overhead due to checkpointing tends to become a minor issue, thus reducing the impact of checkpointing optimizations. Therefore coarse granularities are not considered in this study. On the other hand, finer event granularity will be considered in Section 5.2 when running simulations for a real world model.

As respect to the LP state size, we have varied it between 100 bytes and 8 Kbytes (passing through 2 and 4 Kbytes), so as to cover a relatively large range of values for the state granularity. The results are reported in Figure 5.1.

The plots for the event rate show that semi-asynchronous checkpointing for both \( \mathcal{CA} \) and \( \mathcal{MC} \) provides execution speed improvements over PSS (up to 17% for 8 Kbytes state size) except for the case of minimal state granularity,
i.e. 100 bytes state. Such a result is an expected one since minimal state granularity implies negligible overhead in case of CPU charged checkpointing, which can be comparable with the overhead due to the management of semi-asynchronous checkpointing (e.g. the overhead due to activations of semi-asynchronous checkpointing operations, which involve data exchange between host and LANai memory passing through the PCI bridge).

In addition, a comparison between the event rate of \( \mathcal{M} \) and that obtained for the best threshold of \( \mathcal{C}\mathcal{A} \) shows that the more advanced semantic, namely \( \mathcal{M} \), exhibits no performance increase over \( \mathcal{C}\mathcal{A} \) for small state sizes, while such an increase can be observed in case of medium/large states. An analysis of the values the measured parameters explains the reason. Specifically, in case of state size up to 4 Kbytes, the \( \mathcal{M} \) semantic exhibits in practice the same parameter values as those observed for \( \mathcal{C}\mathcal{A} \) for the values of threshold that maximize the performance.

Instead, for states of 8 Kbytes the situation is different. Specifically, values for F1 are the same for both \( \mathcal{M} \) and \( \mathcal{C}\mathcal{A} \) for the threshold value that maximizes the performance, namely 0.4. However a divergent behavior can be observed for F2. While for threshold set to 0.4, \( \mathcal{C}\mathcal{A} \) exhibits a F2 of 0, a positive value for \( \mathcal{M} \) is observed. This means that, differently from \( \mathcal{C}\mathcal{A} \), \( \mathcal{M} \) aborts some of the checkpoints at the re-synchronization point in line 11 of the engine, thus allowing a reduction of the checkpointing overhead, but only whenever the expected cost of committing a checkpoint is really greater than the expected state recovery cost for that re-synchronization point, thus paying in practice no relevant cost from the point of view of state recovery. This divergence is reflected on a gain of about 500 events per second of \( \mathcal{M} \) over \( \mathcal{C}\mathcal{A} \).

We have noted no relevant difference in the execution efficiency, evaluated as the ratio between the amount of committed events and the total number of executed events (committed plus rolled back), between semi-asynchronous checkpointing and PSS. Therefore the performance gain of semi-asynchronous checkpointing does not derive from reductions of the amount of rollback, i.e. from indirect effects due to changes in the execution mode of the checkpointing protocol. Actually the efficiency values for this experiment are in the order of 85%, with rollback frequency and rollback length in the order of 12% and 1.25 events, respectively.

Another interesting point is that plots for the \( \mathcal{C}\mathcal{A} \) event rate assume different shapes depending on the size of the state. In particular, plots for 100 bytes and 2 Kbytes state size are almost flat. Instead, those for 4 Kbytes and 8 Kbytes state size exhibit a clear dependency on threshold. Specifically, for 4 Kbytes state size the event rate increases from about 12700 to 13300 committed events per sec. while moving threshold from 0.0 to 0.6, and then stabilizes. For 8 Kbytes state size, the event rate starts from about 11700
committed events per sec. when threshold is set to 0.0, and then reaches a peak of about 12900 committed events per sec. when threshold is set to 0.4. Notice that after reaching the maximum value, the curve of the event rate does not stay flat as for the case of 4 Kbytes state size. By these results the low sensitivity of the execution speed vs threshold for the case of relatively small states is an empirical support to the effectiveness of the tracking mechanism for the checkpoint operation advancement described in Section 4.2.5.1. (Recall that such a mechanism underlies also the implementation of $M_C$, which exploits the values of the counters total_transfers and completed_transfers to determine the expected residual completion latency of a checkpointing operation upon re-synchronization). As already said, the mechanism exhibits finer granularity for medium/large states, case in which the selection of threshold, and thus adequate management of the CCA semantic, is a critical factor for the final performance.

An interesting observation is related to the particular shape of the curve for the average distance between committed checkpoints achieved with CCA in case of 8 Kbytes state size. In particular, this curve shows two different steps, one for threshold set to 0.2, the other one for threshold set to 1.0. However, only the first step produces an increase in the event rate due to a decrease in the checkpointing overhead (i.e. the freezing overhead). The reason for this is as follows. The plot related to F1 shows that when threshold is moved from 0.0 to at least 0.2, we get a non-minimal amount of checkpoint aborts due to re-synchronization in line 4. This allows aborting most of the checkpointing operations associated with calls to semi_async_ckpt() to be issued in line 11 of the simulation engine in case the calling LP is re-scheduled for execution, which produces a strong reduction of the checkpointing overhead since committing those checkpoints would be costly due to their recent activation. At the same time, the second step in the average distance between checkpoints is exclusively due to the increase in the frequency F2 when threshold is moved from 0.8 to 1.0, which, unlike the previous case, does not produce relevant decrease in the checkpointing overhead since the additional aborts involve checkpoints with short expected completion latency due to their non-recent activation. Combining this with the monotonic increase of coasting forward cost vs threshold (due to the increase in the average distance between committed checkpoints) we get the slight decrease in the event rate noted after the peak in 0.4.

5.1.2.2 Effects of Increased Workload

In this section we consider a PHOLD configuration similar to the one in Section 5.1.2.1, but with message population increased to 10 messages per LP. As compared to 1 message per LP, this increased workload actually produces
5.1. BENCHMARK STUDY

an execution with longer rollback length (in the order of 2.5 events) occurring with reduced frequency in the order of 3%. In other words, this configuration allows us to test the effects of variations of the rollback pattern in the parallel execution. Note that longer rollback length means that longer antimegasense communication bursts occur during a rollback phase. Therefore this configuration provides indications for communication traffic exhibiting different characteristics. Also in this case we have used 4 machines, with even distribution of the 32 LPs on the machines.

The results are reported in Figure 5.2. They are mostly aligned with those obtained in case of the 1 message per LP workload. There are anyway some differences. For minimal (100 bytes) state size, PSS exhibits a slightly higher gain over semi-asynchronous checkpointing. Also, the performance gain of semi-asynchronous checkpointing for medium/large state size is slightly reduced. This is due to the fact that with a reduced rollback frequency, PSS achieves the best performance for larger value of the checkpoint interval \( \chi \), thus allowing a stronger reduction of the overhead due to synchronous execution of the checkpointing protocol.

As a last observation, differently from the 1 message per LP case, the plot for the CCA event rate stays flat after reaching the maximum value for threshold set to 0.4. This is because the increase in the rollback cost due to the increase in the average distance between committed checkpoints while moving threshold from 0.4 to 1.0 (which produces longer coasting forward) has negligible impact since rollbacks are less frequent.

5.1.2.3 Effects of Increased System Bus Traffic

The benchmark configurations used in Section 5.1.2.1 and in Section 5.1.2.2 are both characterized by event routine implemented as a simple busy loop. In this situation, event execution does not involve main memory access, which does not reduce the system bus bandwidth available for data transfer operations associated with semi-asynchronous checkpointing. To study a more realistic situation in which event execution competes for system bus bandwidth with data transfer associated with semi-asynchronous checkpointing we have considered the same benchmark as the one in Section 5.1.2.1, with a modified structure of the event routine code. Specifically, we have structured the event routine as a loop that performs \texttt{memset}() calls circularly issued on contiguous sets of entries of the state of the LP. This originates RAM accesses for (i) memory updating due to write back from the cache each time contention on a cache slot occurs and (ii) cache updating upon write misses. Note that this configuration is actually an unfavorable test case since event execution code exhibits unrealistically low locality due to the fact that a state entry is referenced only after all the other entries have been referenced by the event routine
software. In addition this setting is expected to be more unfavorable for $\mathcal{MC}$ than for $\mathcal{CCA}$ since $\mathcal{MC}$, based on expected execution costs, behaves better when the event duration does not exhibit large variance. Differently from the previously considered configurations, the event duration is conditioned by the instantaneous system bus load which greatly enhances such a variance. We set the length of the loop performing `memset()` to obtain loop completion within about 150 microseconds when there is no other active user load. As for previous test cases, we have used 4 machines, with even distribution of the LPs on the machines.

The results, reported in Figure 5.3, confirm the tendencies noted for the case of the experiment in Section 5.1.2.1. In particular, the gain due to semi-asynchronous checkpointing over PSS in case of non-minimal state size is confirmed also with this setting. For 8 Kbytes state size, this gain is in the order of 16% when the $\mathcal{MC}$ semantic is adopted and in the order of 13% when $\mathcal{CCA}$ is used.

The only relevant difference observed is in the case of 4 Kbytes state size, which confirms that this setting is less favorable to $\mathcal{MC}$ than $\mathcal{CCA}$. Specifically, instead of exhibiting a slight gain over $\mathcal{CCA}$ as in the previous experiments, the $\mathcal{MC}$ semantic behaves like the optimal $\mathcal{CCA}$ whose gain is in the order of 9% over PSS when threshold is set to 0.6.

### 5.1.2.4 Checkpointing vs Rollback Thrashing and Fossil Collection

As discussed in Section 2.1.4 of Chapter 2, very long sequences of uncheckpointed events (in the order of several tens of events) are undesirable due to possible effects on rollback increase and to possible interactions with the memory recovery procedures, namely fossil collection [31,71,80]. Depending on execution dynamics, a-posteriori checkpoint skipping underlying the $\mathcal{CCA}$ and $\mathcal{MC}$ semantics for re-synchronization might theoretically produce on an LP very long sequences of event executions with no committed checkpoint available for recovery purposes. This might happen especially for large values of `threshold` in $\mathcal{CA}$ (recall that when `threshold` is set to one, $\mathcal{CCA}$ boils down to the $\mathcal{CA}$ semantic presented in Section 3.4.2 of Chapter 3, whose main drawback is just the lack of control on the distance between committed checkpoints at the same LP), or when the `DEFAULT` option is always selected for $\mathcal{MC}$.

To provide hints on the previous issue, we report in Figure 5.4 plots related to the maximum observed distance between consecutive committed checkpoints at an LP in the two benchmarks used in Section 5.1.2.1 and in Section 5.1.2.2. The values are reported for both the $\mathcal{CCA}$ and $\mathcal{MC}$ semantics. The plots show that in $\mathcal{CCA}$, independently of the value of `threshold`, the maximum distance between committed checkpoints is less than 10 in case of 1 message per LP, and is less than 13 in case of 10 messages per LP. On the
other hand the same values for $\mathcal{MC}$ are even lower, less then 8 in case of 1 message per LP and less than 10 in case of 10 messages per LP. This behavior is explained by observing that $\mathcal{MC}$, differently from $\mathcal{CCA}$, keeps memory of the previously uncheckpointed states and is more likely to commit a checkpoint when the distance from the last committed checkpoint increases. Therefore $\mathcal{MC}$, more than $\mathcal{CCA}$ is expected not to give rise to pathological situations with excessively long sequences of uncheckpointed event executions.

However, independently of the used re-synchronization semantic, embedding simple, additional mechanisms in the simulation engine for safety purposes, in order to avoid the maximum distance between committed checkpoints overstepping a given bound (as typically happens when using most CPU charged, adaptive, sparse state saving methods based on a-priori checkpoint skipping [31, 76, 80]) is actually an easy task. This can be accomplished by imposing the value 0 to the parameter threshold of $\mathcal{CCA}$ when the distance from the last committed checkpoint oversteps the bound, or using the $\text{ALWAYS\_FREEZE}$ option characterizing the implementation of $\mathcal{MC}$ in the same situation.

5.1.2.5 Hints on Speedup Values

Speedup, namely the ratio between parallel execution time and serial execution time of the same simulation model, is another relevant measure to understand the performance of semi-asynchronous checkpointing, since it is an indicator of the effectiveness of the parallel execution. By observing this measure we want to be certain whether the gain granted by the semi-asynchronous checkpointing mode is obtained in a condition where the parallel simulation is implemented in a way to be really effective. As respect to this point, speedup in the order of 50% of the ideal one (\footnote{The value $N$ is the ideal speedup when using $N$ machines.}) is typically considered as a threshold to determine acceptable performance for parallel execution on a distributed memory system.

In Figure 5.5 we report the speedup of the parallel simulation for the two benchmarks described in Section 5.1.2.1 and in Section 5.1.2.2, for each of the tested state sizes (100 bytes, 2 Kbytes, 4 Kbytes, 8 Kbytes). The values are reported for both the $\mathcal{MC}$ semantic and $\mathcal{CCA}$ semantic with optimal threshold. The speedup of PSS in the optimal value for $\chi$ is also reported. The plots show that in case of semi-asynchronous checkpointing the speedup is always at least on the order of 65% of the ideal one. In case of increased workload it is in the order of 85%.

Another interesting observation is related to the shape of the plots of semi-asynchronous checkpointing as compared to the plots of the optimal PSS. Specifically, we note how the plots of semi-asynchronous checkpointing are generally flatter than the plot of PSS, thus hinting that semi-asynchronous
checkpointing is effectively removing the greatest part of the checkpointing overhead without paying as much recovery overhead as paid by PSS. This allows similar execution dynamics independently of the LP state size when semi-asynchronous checkpointing is adopted.

5.1.2.6 A Tuning Mechanism for the Value of Threshold in \textit{CCA}

We have seen how the \textit{CCA} semantic, in some circumstances, exhibits an execution speed, namely the event rate, (almost) independent of the selected values for threshold. We have also seen how in other circumstances varying the value of threshold actually determines strong variations in the execution speed. In general, the identification of a well suited value to be assigned to the parameter \text{threshold} (i.e. a value that is likely to maximize the execution speed) should be done in an application specific manner. As a consequence \text{threshold} belongs to the class of the so called \textit{tunable parameters}, whose value should be tuned as the simulation execution proceeds. Those parameters are typically classified along two dimensions [86], namely static vs dynamic and local vs global, and a number of algorithms/heuristics exist in the literature to cope with the tuning problem of those parameters. The basic principle underlying any of these algorithms/heuristics is to tune a parameter value on the basis of the monitoring of a proper reference metric. If the parameter is a global one, the metric the tuning relies on is a global one as well.

From Section 5.1.2.1 to Section 5.1.2.4 we have treated \text{threshold} as a global parameter \footnote{In our context “global” means that its value is the same for all the LPs.}. We improve our perspective in this section by associating with each \textit{LP} a proper value for \text{threshold}, namely \text{threshold}_j, thus allowing the LPs to achieve a good tradeoff between checkpointing overhead (due to freezing associated with committed checkpoints of that LP) and coasting forward overhead (due to uncommitment of checkpoints of that LP) independently of each other as in classical sparse state saving methods based on CPU charged checkpointing [31, 76, 80, 83].

Actually, the heuristic presented in [31] for the adaptive selection of the checkpoint interval \(\chi\) in a PSS method, and then re-used in [73] for the adaptive selection of a threshold value of the simulation time advancement in order to determine the positions of checkpoints on the basis of the event execution pattern in simulation time, can be re-used also to perform adaptive tuning of the parameter \text{threshold}_j in the \textit{CCA} semantic. As pointed out in Section 2.1.1 of Chapter 2, that heuristic is based on the on-line observation of a checkpointing/recovery cost function for each \textit{LP}, namely \(F_j = C_j^{\text{cpt}} + C_j^{\text{f}}\), where \(C_j^{\text{cpt}}\) and \(C_j^{\text{f}}\) represent, respectively, the overheads due to checkpointing and coasting forward for \textit{LP}_j. Specifically, in the original solution [31], the func-
tion $F_j$ is monitored over successive observation periods and the checkpoint interval $\chi$ is increased/decreased on the basis of the monitored variations.

In our context, the algorithm should work as follows. At the first observation period threshold$_j$ is set to zero; in the successive observation periods threshold$_j$ is increased by a quantity $\epsilon$ (say 0.1) if $F_c$ did not increase. Otherwise the adaptation direction of threshold$_j$ is inverted (threshold$_j$ is decreased by $\epsilon$). The inversion of the adaptation direction takes place each time the last observed value of the cost function is greater than the previous one. So, the idea underlying the recalculation is to avoid as much freezing overhead as possible (by increasing threshold$_j$, which leads to the commitment of only those checkpoints of LP$_j$ with reduced expected completion latency) until the overhead reduction is over-stepped by the increasing cost caused by uncommitment of checkpoints.

To test adaptive tuning we have used a modification of the PHOLD benchmark used in Section 5.1.2.1. In this test case, there are 4 hot spot LPs to which the 30% of all the messages are routed. The hot spot LPs change in the course of the simulation according to a sequence of changes which is defined prior to the simulation execution by randomly picking up new hot spots among all the 32 LPs, with the constraint that at least one hot spot LP per each of the 4 used machines is ensured. This configuration possibly gives rise to simulations which do not reach steady state of the rollback behavior of the LPs. Also, different rollback patterns might arise on different LPs thus yielding a situation in which different threshold selections might actually provide benefits. We have selected 8 Kbytes as the state size in order to provide results for a large state granularity application, instance in which performance benefits due to adequate tuning of the parameter threshold should be more evident (3).

We report in Figure 5.6 the event rate vs the value of threshold in case of manual selection (we recall that in this case the same value is adopted for all the LPs), and also the event rate achieved with the previously described adaptive tuning. The results point out that adaptive tuning performs slightly better than manual selection, which supports the feasibility of re-usage of the heuristic in [31] for solving the tuning problem.

5.1.3 Part B

This part of the study is dedicated to the observation of the effects of combined use of a-posteriori and a-priori checkpoint skipping when the semi-asynchronous execution mode of the checkpointing protocol is used. In this

---

3If the sensitivity of the execution speed vs threshold were relatively low, as for the case of small/medium state granularity, it would be difficult to provide hints on the effectiveness of an adaptive tuning of threshold itself.
section we use the benchmark of Section 5.1.2.1 as the test case (always run on 4 machines). For the \textit{CCA} semantic we report plots for the event rate vs the parameter \texttt{threshold} while adopting a set of different values for the checkpoint interval \( \chi \) of the LPs. For the \textit{MC} semantic we directly report the event rate vs \( \chi \) since the parameter \texttt{threshold} has no sense in \textit{MC}. These plots allow us to observe possible variations in the execution speed in case a checkpoint interval larger than 1 is adopted, i.e. in case semi-asynchronous checkpointing is used in combination with a classical periodic method for placing checkpoints. Also in this case we have fixed the LP state size at 8 Kbytes. We have made this choice since models with small state sizes typically exhibit the best execution performance when the checkpoint interval is set to 1, i.e. when no a-priori skipping is used at all. Therefore, fine state granularity would not allow us to understand whether combined use of a-posteriori and a-priori checkpoint skipping might really yield execution speed improvements.

The plots for the event rate, reported in Figure 5.7 for both \textit{MC} (Figure 5.7) and \textit{CCA} (Figure 5.8) semantics show that the best performance is achieved when using a checkpoint interval set to 2, with an event rate value of about 13600 committed events per sec. in case of \textit{MC} and 13100 committed events per sec. in case of \textit{CCA}. Note however that independently on the re-synchronization semantic employed, the configuration with checkpoint interval set to 1, i.e. with no a-priori checkpoint skipping, achieves almost the same execution speed as the configuration with checkpoint interval set to 2, while larger values of the checkpoint interval produce worse event rate values. Overall, the use of pure a-posteriori skipping (i.e. checkpoint interval set to 1) exhibits performance very close to the best one achievable with combined use of a-priori and a-posteriori skipping. Similar results have been observed for the benchmarks of Section 5.1.2.2 and of Section 5.1.2.3. This points out the effectiveness of a-posteriori checkpoint skipping even if used alone.

A final interesting point relates to the shape of the event rate curves in case the \textit{CCA} semantic is employed. For checkpoint interval set to 2, the event rate does not vary while moving \texttt{threshold} from 0.0 to 1.0. A similar behavior is noted for checkpoint interval set to 3 and \texttt{threshold} in the interval between 0.0 and 0.8. Instead, for any other value of the checkpoint interval, the event rate curve exhibits a clear dependency on the value of \texttt{threshold}. Specifically, for checkpoint interval set to 4 or 5, the event rate monotonically decreases while increasing \texttt{threshold}. On the other hand, for checkpoint interval set to 1, the event rate exhibits the peak value for an intermediate value of \texttt{threshold}. Overall, the only configuration that takes advantage while adopting adequate values for \texttt{threshold} is the one with checkpoint interval set to 1. The reason for this behavior is as follows. When a checkpoint interval larger than 1 is used, skipping some checkpoints due to checkpoint abort does not produce strong reduction in the checkpointing overhead since
fewer activations of the checkpointing protocol occur within a given execution time period. On the other hand, it might produce a non-minimal increase in the distance between committed checkpoints at the same LP, with consequent non-minimal increase of the expected rollback overhead due to non-minimal increase in the expected coasting forward length. Actually the two overheads get balanced for relatively small checkpoint intervals (i.e. 2 or 3). Instead, the increased cost of rollback due to checkpoint aborts dominates when larger checkpoint intervals are adopted.

To support the previous deduction, we report in Figure 5.9 the cumulated overhead per executed event due to both checkpointing and coasting forward when CC.A is adopted. By the plots we see that the cumulated overhead does not vary with threshold for checkpoint interval set to 2 and 3. (For the latter value it exhibits an increase only while moving threshold from 0.8 to 1.0. This increase has the effect of producing the slight decrease in the event rate shown in Figure 5.7.) It increases vs threshold for checkpoint interval larger than 3. Instead it exhibits a minimum vs threshold for the case of checkpoint interval set to 1, which justifies the peak appearing in the corresponding event rate curve.

5.2 Study on a Real World Model

In this section we report results demonstrating the benefits from semi-asynchronous checkpointing for optimistic parallel simulations of a Personal Communication System (PCS) model. Besides allowing testing of semi-asynchronous checkpointing on a real world application, this section definitely enlarges the spectrum of test cases considered in this dissertation since the event granularity for the specific PCS simulation model is much finer as respect to the event granularity considered in the previous experiments.

As a testing environment we have used the same one employed in Section 5.1, with the only difference that a larger number of machines, namely 8 machines, have been used. Also, similarly to what was done in Section 5.1, periodic state saving (PSS) has been used as the reference checkpointing technique based on synchronous (CPU charged) execution of the checkpointing protocol.

A PCS provides communication services to mobile units. In our simulation model the service area is partitioned into cells, each of which is modeled by a distinct LP. Each cell represents a receiver/transmitter having either some fixed number of channels allocated to it (Fixed-Channel-Assignment, namely FCA) or a number of channels dynamically assigned to it (Dynamic-Channel-Assignment, namely DCA). In this paper we consider an FCA model with 100 channels per cell. The model is call-initiated [14] since it only simulates the
behavior of a mobile unit during conversation (i.e. the movement of a mobile unit is not tracked unless the unit itself is in conversation). Therefore, the model is organized around two entities, namely cells and calls. Call requests arrive to each cell according to an exponential distribution [8, 11, 14] with inter-arrival time 10 seconds. All the calls initiated within a given cell are originated by the LP associated with that cell, therefore no external call generator is used. There are three main types of events, namely hand-off (due to mobile unit cell switch), call termination and call arrival. When a call arrives at a cell, channel availability must be determined. If all channels are busy, the incoming call is simply counted as a “block”. If at least one channel is available, then channel assignment for the new call takes place. A call termination simply involves the release of the associated channel and statistics update.

Hand-off takes place each time a mobile unit currently involved in conversation moves from one cell to another. In our model there are two distinct classes of mobile units. Both of them are characterized by a residence time within a cell which follows an exponential distribution, with mean 5 minutes (fast movement units) and 40 minutes (slow movement units), respectively. The average holding time for each call associated with both fast and slow movement units is 2 minutes. When a call arrives at a cell, the type (slow or fast) of the mobile unit associated with the incoming call is selected from a uniform distribution, therefore any call is equally likely to be destined to a fast or a slow movement mobile unit. When a hand-off occurs between adjacent cells, the hand-off event at the cell left by the mobile simply involves the release of the channel. Instead, the hand-off event at the destination cell checks for channel availability. If there is no available channel, then the call is simply cut off (dropped), otherwise an available channel is assigned to the call. Hand-off events for destination cells are not pre-computed, i.e. they are scheduled only upon the occurrence of the hand-off event at the cell left by the mobile unit.

The state of each LP records statistics, information about busy channels and, for each channel, information about features of the mobile unit involved in the ongoing call (e.g. scheduled call termination time, call initiation time, class of the mobile unit etc.), if any. As a result, the size of the state is about 4Kbytes \(^4\). Also, the event granularity for this model is in the order of 15/20 microseconds. We have simulated a PCS in which each cell is hexagonal, therefore all the cells, except bordering cells of the coverage area, have six neighbor cells. The model size, in terms of number of cells, has been fixed at 64. The corresponding 64 LPs have been evenly distributed among the 8

\(^4\)PCS models might exhibit smaller LP state granularity [11, 14]. This might happen when information maintained in the LP state are almost exclusively related to the current state of the channels within the associated cell, or when the number of channels per cell is small.
machines of the cluster.

We report in Figure 5.10 the event rate vs threshold when adopting semi-asynchronous checkpointing (CCA re-synchronization semantic) in combination with no a-priori checkpoint skipping. For this experiment we have considered threshold as a global parameter. The abort frequencies F1 and F2 (as defined in Section 5.1) are also reported in Figures 5.11 and 5.12. Figure 5.13 reports the corresponding values in case of semi-asynchronous checkpointing used in combination with the MC re-synchronization semantic. We also report in Figure 5.14 the event rate vs the checkpoint interval $\chi$ of the LPs when adopting the synchronous execution mode of the checkpointing protocol.

The plots indicate that semi-asynchronous checkpointing allows a significant acceleration of the simulation model execution. In particular, the best event rate of semi-asynchronous checkpointing is about 65500 events per sec., achieved for the MC semantic, while the best event rate with synchronous checkpointing is about 57000 events per sec., achieved when the checkpoint interval $\chi$ is set to 8. Thus semi-asynchronous checkpointing provides a performance gain in the order of 14%.

The observed values of F2, both in the case of MC and CCA, lead to an important observation. Specifically, we note that for both semantics F2 is relevantly different from zero, which is a result different from those observed for the PHOLD benchmark. The reason lies on the event granularity for the PCS model (15/20 microseconds) which is much shorter than that used in the PHOLD benchmark (about 150 microseconds). Thus, in the PCS simulation, we get higher likelihood to find a still ongoing semi-asynchronous checkpointing operation upon re-synchronization in line 11 of the simulation engine.

Another observation on the abort frequencies can be done by noting that the F2 values of MC and CCA, with threshold set to 1, are similar, which means that MC discards almost all the uncompleted checkpoints upon re-synchronization in line 11, just like the CCA semantic does. However the performance values exhibit a relevant difference in favor of MC, whatever threshold is adopted for CCA. This is a support that the simpler CCA semantic is not achieving the best tradeoff due to semi-asynchronous checkpoint commit/abort.

As a last remark we would like to bring to the reader's attention, semi-asynchronous checkpointing with CCA provides good performance independently of the selected value for the parameter threshold. Specifically, the minimum observed event rate is about 57500 committed events per sec., achieved when threshold is set to 0.0, which is even slightly better than the best event rate achievable with synchronous execution of the checkpointing protocol.
Figure 5.1: Results for the Basic Test Case.
<table>
<thead>
<tr>
<th>State Size</th>
<th>Event Rate</th>
<th>F1</th>
<th>F2</th>
<th>Av. Dist. Between Committed Ckpts</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 Bytes</td>
<td>15932</td>
<td>0.07</td>
<td>0.00</td>
<td>1.05</td>
</tr>
<tr>
<td>2 Kbytes</td>
<td>15613</td>
<td>0.12</td>
<td>0.00</td>
<td>1.14</td>
</tr>
<tr>
<td>4 Kbytes</td>
<td>15732</td>
<td>0.15</td>
<td>0.00</td>
<td>1.18</td>
</tr>
<tr>
<td>8 Kbytes</td>
<td>15678</td>
<td>0.15</td>
<td>0.04</td>
<td>1.25</td>
</tr>
</tbody>
</table>

**Figure 5.2: Results for Increased Workload.**
Figure 5.3: Results for Increased System Bus Traffic.
Figure 5.4: Maximum Distance Between Committed Checkpoints.

Figure 5.5: Speedup Results.
Figure 5.6: \( CCA \) Semantic - Static vs Adaptive Tuning of Threshold.

Figure 5.7: \( CCA \) semantic - Event Rate vs Threshold for Different Checkpoint Intervals.
Figure 5.8: $\mathcal{M}$C Semantic - Event Rate vs $\chi$.

Figure 5.9: $\mathcal{C}CA$ Semantic - Cumulated Overhead per Executed Event vs Threshold for Different Checkpoint Intervals.
Figure 5.10: PCS Application - Event Rate vs Threshold for Semi-Asynchronous Checkpointing (CCA Semantic).

Figure 5.11: PCS Application - F1 vs Threshold for Semi-Asynchronous Checkpointing (CCA Semantic).
Figure 5.12: PCS Application - F2 vs Threshold for Semi-Asynchronous Checkpointing (\(\mathcal{C}\mathcal{A}\) Semantic).

<table>
<thead>
<tr>
<th>(\mathcal{MC}) Semantic</th>
<th>Event Rate</th>
<th>F1</th>
<th>F2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>65024</td>
<td>0.17</td>
<td>0.29</td>
</tr>
</tbody>
</table>

Figure 5.13: PCS Application - Results for Semi-Asynchronous Checkpointing (\(\mathcal{MC}\) Semantic).

Figure 5.14: PCS Application - Event Rate vs the Checkpoint Interval \(\chi\) for Synchronous Checkpointing.
Chapter 6

Conclusions

This dissertation presents an innovative mode to execute checkpointing, namely semi-asynchronous checkpointing, in support of optimist parallel discrete event simulation. This mode relaxes the widespread assumption that checkpointing operations have to be executed according to the synchronous paradigm, which implies blocking of any other simulation specific operation while checkpointing is in progress.

We have shown that in the context of optimistic parallel discrete event simulation such an assumption is not required and the consequences of its relaxation have been explored. In particular, by abandoning the synchronous paradigm, we allow checkpointing to be carried out in real concurrency with other simulation specific operations, thus aiming at removing the cost of any single checkpointing operation from the completion time of the parallel simulation application.

The presence of real concurrency introduces a set of consistency and contention problems. Those problems have been identified and the notion of re-synchronization has been proposed as a common solution. Four different semantics for re-synchronization have been proposed, from the simple \( \mathcal{AF} \) to the complex \( \mathcal{MC} \), passing through \( \mathcal{CA} \) and \( \mathcal{CCA} \), each one exhibiting a different tradeoff between complexity and efficiency.

In addition, to prove the practical feasibility of this checkpointing mode, the dissertation presents a software library supporting semi-asynchronous checkpointing on myrinet based clusters, discusses its most relevant engineering issues and the relative solutions, and uses it to carry out experiments both on parameterized synthetic benchmarks and on a real world simulation application. In both situations semi-asynchronous checkpointing showed relevant performance improvement whenever the weight of the checkpointing cost on the overall simulation execution time is significant, i.e. when checkpointing really turns out to be a performance critical task, thus revealing itself a cost
effective solution to support state recovery in optimistic parallel simulation.
Bibliography


