

PORTABLE CHECKPOINTING FOR PARALLEL  
APPLICATIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Grigory Bronevetsky

January 2007

© 2007 Grigory Bronevetsky

ALL RIGHTS RESERVED

# PORTABLE CHECKPOINTING FOR PARALLEL APPLICATIONS

Grigory Bronevetsky, Ph.D.

Cornell University 2007

High Performance Computing (HPC) systems represent the peak of modern computational capability. As ever-increasing demands for computational power have fuelled the demand for ever-larger computing systems, modern HPC systems have grown to incorporate hundreds, thousands or as many as 130,000 processors. At these scales, the huge number of individual components in a single system makes the probability that a single component will fail quite high, with today's large HPC systems featuring mean times between failures on the order of hours or a few days. As many modern computational tasks require days or months to complete, fault tolerance becomes critical to HPC system design.

The past three decades have seen significant amounts of research on parallel system fault tolerance. However, as most of it has been either theoretical or has focused on low-level solutions that are embedded into a particular operating system or type of hardware, this work has had little impact on real HPC systems. This thesis attempts to address this lack of impact by describing a high-level approach for implementing checkpoint/restart functionality that decouples the fault tolerance solution from the details of the operating system, system libraries and the hardware and instead connects it to the APIs implemented by the above components. The resulting solution enables applications that use these APIs to become self-checkpointing and self-restarting regardless of the the software/hardware plat-

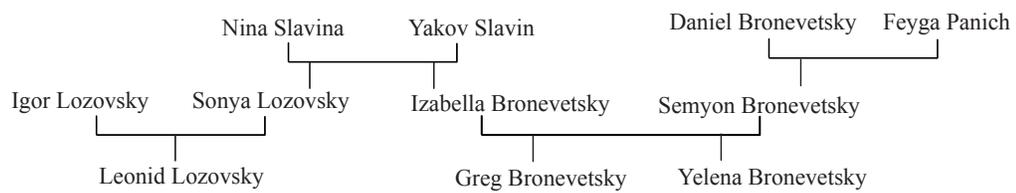
form that may implement the APIs.

The particular focus of this thesis is on the problem of checkpoint/restart of parallel applications. It presents two theoretical checkpointing protocols, one for the message passing communication model and one for the shared memory model. The former is the first protocol to be compatible with application-level checkpointing of individual processes, while the latter is the first protocol that is compatible with arbitrary shared memory models, APIs, implementations and consistency protocols. These checkpointing protocols are used to implement checkpointing systems for applications that use the MPI and OpenMP parallel APIs, respectively, and are first in providing checkpoint/restart to arbitrary implementations of these popular APIs. Both checkpointing systems are extensively evaluated on multiple software/hardware platforms and are shown to feature low overheads.

## **BIOGRAPHICAL SKETCH**

Greg Bronevetsky was born on April 27th 1978 in Kiev, Ukraine. He immigrated to the United States of America in 1989, residing in New York, Pennsylvania and New Jersey. Greg attended Trenton State College (later renamed The College of New Jersey) and graduated summa cum laude in May of 1999 with departmental honors. After working for a year at Charles Jones LLC, in Trenton, NJ, Greg enrolled in the Ph.D. program in the Department of Computer Science at Cornell University in August of 2000. He received a Ph.D. in Computer Science in January of 2007.

This work is dedicated to my family.



## ACKNOWLEDGEMENTS

Many people have contributed to the work that makes up this thesis. I would like to thank and acknowledge their many contributions.

My deepest thanks go to my advisor, Keshav Pingali. His deep insight guided my work to a successful conclusion by making sure from the start that my work was indeed useful and valuable to the wider computer science community. It is from him that I learned to overcome my tendency to work on things that are merely interesting and to also ask the questions "Is this useful?" and "How can my work have a real impact?". Furthermore, Keshav's constant focus on clear communication taught me to present my ideas in the simplest and clearest manner possible. Thanks to him, I have not only become a better speaker and writer, but have also come to a clearer understanding of my own work.

Paul Stodghill has my gratitude for his valuable help as a collaborator. My discussions with Paul have been illuminating both in helping me advance my own thinking and in providing a firm base in practicality and empiricism on which to base our work. I greatly appreciate the time he has given to understanding my suggestions and helping to improve them by making them simpler and more practical. Moreover, Paul's excellent technical and organization skills have kept myself and others productive both individually and as a team.

I would like to thank Sally McKee both for our research discussions and for her valuable help in my job search. She has been my friend and a cheerleader for my work.

My thanks go to José Martínez, Jon Kleinberg and Anil Nerode for being on my committee and helping to guide my work.

I have nothing but appreciation and gratitude to my colleagues James Ezick, Rohit Fernades, Milind Kulkarni, Daniel Marques, Martin Shulz and Kamen Yotov. I have benefitted greatly from their work, ideas and friendship.

I thank Cornell University, its Department of Computer Science, its Computer Systems Laboratory and the many professors therein whose courses I have taken and whose knowledge I have tried to absorb.

I thank my professors from The College of New Jersey, Penny Anderson, Roy Clouser, Charles Goldberg, Yong Lee, Deborah Knox, Norman Neff and Ursula Wolz. In addition to providing me with a solid foundation in Computer Science and critical reasoning, they have helped me to expand my abilities and horizons though class work as well as industrial and research experience.

Among the many fine teachers who influenced my early development I would like to offer special thanks to Rabbi Dov Kagan and Mr. David Weinstein of Sinai Academy. Rabbi Kagan's creative teaching style, tireless dedication to his work and his endless patience with his students had a remarkable effect on me, helping me to become both a better student and a better person. Mr. Weinstein's teaching talent and patience with a loud but promising boy sparked my interest in computers and gave me a big push towards where I am today. I thank you both.

Little would have happened during my stay at Cornell without the work of Becky Stewart, Stephanie Meik and other members of the Cornell Computer Science administrative staff. I would like to thank them for clearing my path through Cornell, constantly trying to make my life easier and simpler even when that wasn't an easy thing to do.

My family, to whom this thesis is dedicated, has been a constant support and comfort in my life and I would like to thank them both for bringing me into this

world and for guiding and supporting me on my path through it. Everything I do is thanks to you and whatever I may accomplish in life, my debt to you will only grow. While nothing I do can repay it, I can at least thank you and do my best to make you proud.

Finally, I would like to express my gratitude and thanks to Raga Krishnakumar, whose love and friendship have been invaluable to me. You have made my life brighter, warmer and more complete. All that I can aspire to is to give the same to you.

## TABLE OF CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Rollback Restart . . . . .	3
1.1.1	Basic Checkpointing Techniques . . . . .	3
1.1.2	The Four R's of Checkpointing . . . . .	5
1.1.3	Current State of the Art . . . . .	7
1.1.4	Practical Checkpointer Design . . . . .	10
1.2	Organization of Thesis . . . . .	13
<b>2</b>	<b>Checkpointing Message Passing Applications</b>	<b>14</b>
2.1	Background . . . . .	14
2.2	Taxonomy of Solutions . . . . .	21
2.2.1	Applying the Taxonomy . . . . .	23
2.2.1.1	Checkpointing . . . . .	23
2.2.1.2	Message Logging . . . . .	30
2.3	Checkpointing Protocols . . . . .	31
2.3.1	Communication-free protocols . . . . .	32
2.3.1.1	Application-Aware Protocols . . . . .	33
2.3.1.2	Automatic Protocols . . . . .	35
2.3.2	Uncoordinated Checkpointing . . . . .	36
2.3.3	Quasi-Synchronous Checkpointing . . . . .	40
2.3.3.1	Strictly Z-Path Free . . . . .	44
2.3.3.2	Z-Path Free . . . . .	44
2.3.3.3	Z-Cycle Free . . . . .	45
2.3.3.4	Performance Considerations . . . . .	46
2.3.4	Coordinated Checkpointing . . . . .	47
2.3.4.1	Blocking Coordinated Checkpointing . . . . .	48
2.3.4.1.1	Sync-and-stop . . . . .	48
2.3.4.1.2	Time-coordinated . . . . .	50
2.3.4.2	Non-Blocking Coordinated Checkpointing . . . . .	53
2.3.4.2.1	Late-crossed Restart Lines . . . . .	53
2.3.4.2.2	Criss-crossed Restart Lines . . . . .	57
2.4	Message Logging Protocols . . . . .	58
2.4.1	Pessimistic . . . . .	59
2.4.2	Optimistic . . . . .	62
2.4.3	Causal . . . . .	66
2.5	Rollback Restart for MPI Applications . . . . .	68
2.5.1	The MPI Specification . . . . .	69
2.5.1.1	Point-to-point Communication . . . . .	70
2.5.1.2	Collective Communication . . . . .	72
2.5.1.3	Opaque Objects . . . . .	75
2.5.2	Prior Work on Rollback Restart . . . . .	78

2.5.2.1	Manual Solutions . . . . .	78
2.5.2.2	Automated Implementation-specific Solutions . . .	80
2.5.2.3	Automated Implementation-independent Solutions	83
2.5.3	RROMP . . . . .	90
2.5.4	Checkpointing Protocol . . . . .	92
2.5.4.1	High-level description of protocol . . . . .	93
2.5.4.2	Piggybacked information on messages . . . . .	95
2.5.4.3	Completion of receipt of late messages . . . . .	98
2.5.4.4	Putting it all together . . . . .	99
2.5.4.5	Restart . . . . .	100
2.5.5	Blocking Point-to-Point Communication . . . . .	103
2.5.6	Non-blocking Point-to-Point Communication . . . . .	104
2.5.6.1	Applying the Protocol . . . . .	104
2.5.6.2	Non-determinism Issues . . . . .	106
2.5.7	Non-deterministic Message Arrival . . . . .	107
2.5.8	Collective Communication . . . . .	108
2.5.8.1	Difficulties . . . . .	109
2.5.8.2	Semantics of Collectives . . . . .	111
2.5.8.3	Integrating Collectives into RROMP . . . . .	113
2.5.8.4	All-to-all collective operations . . . . .	114
2.5.8.5	Single-receiver collective operations . . . . .	117
2.5.8.6	Single-sender collective operations . . . . .	121
2.5.8.7	Barriers . . . . .	122
2.5.8.8	Restart . . . . .	125
2.5.9	MPI Opaque State . . . . .	126
2.5.9.1	Datatypes, Communicators and Groups . . . . .	127
2.5.9.2	Reduction Operations . . . . .	131
2.5.10	Experimental Evaluation . . . . .	133
2.5.10.1	Small-scale Applications Experiment . . . . .	134
2.5.10.1.1	Experimental setup . . . . .	134
2.5.10.1.2	RROMP Overheads . . . . .	136
2.5.10.2	Large-scale Applications Experiment . . . . .	138
2.5.10.2.1	Experimental Setup . . . . .	140
2.5.10.2.2	Overhead Without Checkpoints . . . . .	142
2.5.10.2.3	Overhead With Checkpoints . . . . .	145
2.5.10.2.4	Restart Cost . . . . .	149
2.5.10.2.5	Discussion . . . . .	149
2.5.10.3	Piggybacking Overhead . . . . .	151
2.5.10.3.1	Point-to-point Piggybacking . . . . .	154
2.5.10.3.2	Collective Piggybacking . . . . .	160
2.5.10.3.3	Piggybacking and Application Performance	164
2.5.10.3.4	Discussion . . . . .	167

<b>3</b>	<b>Parallel Checkpointing - Shared Memory</b>	<b>169</b>
3.1	Background . . . . .	169
3.1.1	Shared Memory Models . . . . .	171
3.2	Prior Work . . . . .	174
3.2.1	Coordinated Checkpointing . . . . .	177
3.2.2	Uncoordinated Checkpointing . . . . .	181
3.2.3	Quasi-Synchronous Checkpointing . . . . .	181
3.2.4	Message Logging . . . . .	183
3.2.5	Miscellaneous . . . . .	186
3.3	Availability of Rollback Restart . . . . .	187
3.4	Portable Checkpointing Protocol . . . . .	190
3.4.1	Approach . . . . .	190
3.4.2	Synchronization and the Protocol . . . . .	191
3.4.3	Refining the Basic Protocol . . . . .	193
3.4.4	Incorporating Locks . . . . .	197
3.4.5	Incorporating Semaphores . . . . .	201
3.4.6	Incorporating Barriers . . . . .	204
3.4.7	Implicit Synchronization . . . . .	206
3.4.8	Eager vs. Lazy . . . . .	206
3.5	Applying the Protocol to OpenMP . . . . .	207
3.5.1	Architecture . . . . .	209
3.5.2	State classification . . . . .	211
3.5.3	The 4R's Applied to OpenMP . . . . .	212
3.5.4	Discussion . . . . .	213
3.6	Hidden State . . . . .	214
3.6.1	Threads . . . . .	214
3.6.1.1	Recreate threads and their IDs . . . . .	214
3.6.1.2	Recreate thread to stack mapping . . . . .	215
3.6.1.3	Recreate the stack contents . . . . .	216
3.6.2	Privatized and Reduction Variables . . . . .	218
3.6.3	Worksharing constructs . . . . .	218
3.7	Synchronization State . . . . .	222
3.7.1	Potential Checkpoint Locations . . . . .	222
3.7.2	Incorporating OpenMP Synchronization Constructs . . . . .	223
3.7.2.1	Incorporating Atomic Updates . . . . .	223
3.7.2.2	Incorporating Critical Regions . . . . .	224
3.7.2.3	Incorporating Application-Implemented Synchroniza- tion . . . . .	227
3.8	Details of transformations and protocols . . . . .	228
3.8.1	Overview . . . . .	228
3.8.2	Transformations . . . . .	229
3.8.2.1	Transformation #1: Directive body lifting . . . . .	229
3.8.2.2	Transformation #2: Parallel For Normal Form . . . . .	232
3.8.2.3	Transformation #3: Barrier Replacement . . . . .	233

3.8.2.4	Transformation #4: Privatization . . . . .	236
3.8.2.5	Transformation #5: Recreation of the stack and the threads . . . . .	244
3.8.2.6	Transformation #6: Stack Alignment . . . . .	251
3.8.2.6.1	Stack Motion . . . . .	252
3.8.2.6.2	Stack region size changes . . . . .	255
3.8.2.7	Transformation #7: Saving state of local and global variables . . . . .	256
3.8.2.8	Transformation #8: Saving the state of the heap .	260
3.8.3	Checkpointing Protocol . . . . .	261
3.8.3.1	Protocol state . . . . .	261
3.8.3.1.1	Shared variables . . . . .	261
3.8.3.1.2	Thread-private variables . . . . .	262
3.8.3.2	Parallel regions . . . . .	264
3.8.3.3	Master . . . . .	267
3.8.3.4	Atomic . . . . .	267
3.8.3.5	Potential and explicit checkpoints . . . . .	268
3.8.3.6	Locks . . . . .	276
3.8.3.7	Barriers . . . . .	277
3.8.3.8	Critical regions . . . . .	280
3.8.4	Worksharing Constructs . . . . .	286
3.8.4.1	Ordered . . . . .	302
3.8.4.2	Critical and Ordered . . . . .	324
3.8.5	Miscellany . . . . .	325
3.8.5.1	Finer-grained coordination. . . . .	325
3.8.5.2	Environment routines . . . . .	326
3.9	Implementation and Experiments . . . . .	327
3.9.1	Application Overheads - SPLASH-2 . . . . .	329
3.9.1.1	Checkpoint-free Overheads . . . . .	329
3.9.1.2	Checkpoint and Restart Overhead . . . . .	331
3.9.1.3	Checkpoint Size Comparison to SLC . . . . .	335
3.9.2	Application Overheads - NAS . . . . .	338
3.9.2.1	Checkpoint-free Overheads . . . . .	338
3.9.2.2	Cost of Checkpointing . . . . .	344
3.9.2.3	Cost of Restarting . . . . .	348
3.9.2.4	Checkpoint Size Comparison to SLC . . . . .	352
3.9.3	Detailed Examination of Overheads . . . . .	353
3.9.3.1	Synchronization . . . . .	354
3.9.3.2	Scheduling . . . . .	358
3.9.3.3	Array . . . . .	359
3.9.4	Discussion . . . . .	367
3.10	Summary . . . . .	368

<b>4</b>	<b>Future Work</b>	<b>371</b>
4.1	Generic Rollback Rollback Restart for MPI . . . . .	371
4.2	Checkpointing Hybrid MPI/OpenMP Applications . . . . .	373
4.3	Hybrid Checkpointing/Message Logging . . . . .	374
4.4	Soft Error Vulnerability Compiler Analysis . . . . .	377
4.4.1	Moving Forwards . . . . .	379
4.4.2	Fault Model . . . . .	381
4.4.3	Compiler Analysis . . . . .	381
4.4.4	Guided Fault Tolerance . . . . .	384
	<b>Bibliography</b>	<b>386</b>

## LIST OF TABLES

2.1	RROMP, Checkpointing overhead; running times and relative % overheads	137
2.2	Runtimes in seconds on Lemieux without checkpoints . . . . .	143
2.3	Runtimes in seconds on Velocity 2 without checkpoints . . . . .	144
2.4	Runtimes in seconds on Lemieux with checkpoints . . . . .	146
2.5	Runtimes in seconds on Velocity 2 with checkpoints . . . . .	147
2.6	Restart costs in seconds on Lemieux . . . . .	150
2.7	Restart costs in seconds on CMI . . . . .	150
3.1	Characteristics of SPLASH-2 Benchmarks . . . . .	330
3.2	Benchmark input sizes . . . . .	331
3.3	SPLASH-2 Linux/Athlon Experiments . . . . .	332
3.4	SPLASH-2 Tru64/Alpha Experiments . . . . .	333
3.5	Overhead of Checkpoint and Restart on Linux/Athlon. . . . .	335
3.6	Overhead of Checkpoint and Restart on Tru64/Alpha. . . . .	336
3.7	SPLASH-2 Linux/Athlon $C^3$ vs BLCR Checkpoint Sizes . . . . .	337
3.8	NAS Linux/IA64 Original Runtimes, No-Checkpoint Overheads and Checkpoint Sizes (40 runs each) . . . . .	341
3.9	NAS Tru64/Alpha Original Runtimes, No-Checkpoint Overheads and Checkpoint Sizes(35 runs each) . . . . .	342
3.10	NAS Solaris/Sparc Original Runtimes, No-Checkpoint Overheads and Checkpoint Sizes (40 runs each) . . . . .	343
3.11	NAS Linux/IA64 1-Checkpoint Times in Secs and % of Runtime(35 runs each) . . . . .	345
3.12	NAS Tru64/Alpha 1-Checkpoint Times in Secs and % of Runtime(40 runs each) . . . . .	346
3.13	NAS Solaris/Sparc 1-Checkpoint Times in Secs and % of Runtime(40 runs each) . . . . .	347
3.14	NAS Linux/IA64 Restart Times in Secs and % of Runtime(35 runs each)	349
3.15	NAS Tru64/Alpha Restart Times in Secs and % of Runtime(40 runs each)	350
3.16	NAS Solaris/Sparc Restart Times in Secs and % of Runtime(40 runs each)	351
3.17	NAS Linux/Athlon $C^3$ vs BLCR Checkpoint Sizes . . . . .	353
3.18	Linux/IA64 Synchronization Microbenchmarks (in $\mu\text{sec}$ ) (60 runs each)	357
3.19	Tru64/Alpha Synchronization Microbenchmarks (in $\mu\text{sec}$ ) (90 runs each)	357
3.20	Solaris/Sparc Synchronization Microbenchmarks (in $\mu\text{sec}$ ) (60 runs each)	358
3.21	Linux/IA64 Scheduling Microbenchmarks (times in $\mu\text{sec}$ ) . . . . .	360
3.22	Tru64/Alpha Scheduling Microbenchmarks (times in $\mu\text{sec}$ ) (90 runs each)	361
3.23	Solaris/Sparc Scheduling Microbenchmarks (times in $\mu\text{sec}$ ) . . . . .	362
3.24	Linux/IA64 Array Microbenchmarks (times in $\mu\text{sec}$ ) . . . . .	364
3.25	Tru64/Alpha Array Microbenchmarks (times in $\mu\text{sec}$ ) (90 runs each) . .	365
3.26	Solaris/Sparc Array Microbenchmarks (times in $\mu\text{sec}$ ) . . . . .	366

## LIST OF FIGURES

1.1	Sequential system stack . . . . .	4
2.1	Rollback restart layer between the application and the communication system . . . . .	16
2.2	Example restart line . . . . .	16
2.3	Example of checkpoints and restart lines . . . . .	17
2.4	Restart line with Past, Future, Late and Early messages . . . . .	18
2.5	Example of checkpoints and labeled restart lines . . . . .	22
2.6	Code example where all restart lines are crossed by a late message . . . . .	24
2.7	Example of late-crossed restart line creation . . . . .	25
2.8	Time intervals of when early messages may be sent vs when late messages may be sent. . . . .	26
2.9	Code example where all restart lines are crossed by early messages . . . . .	29
2.10	All restart lines crossed by early message . . . . .	29
2.11	Example message logging restart line . . . . .	31
2.12	Example of the domino effect . . . . .	38
2.13	Examples of zig zag and causal paths . . . . .	41
2.14	Examples of a zig zag cycle . . . . .	42
2.15	Blocking checkpointing of an application . . . . .	47
2.16	Examples of a blocking coordinated checkpoint . . . . .	49
2.17	Outline of the blocking time-coordinated protocol . . . . .	51
2.18	Outline of the semi-blocking time-coordinated protocol . . . . .	52
2.19	Limits on when checkpoints must be taken . . . . .	54
2.20	Sample message logging restart lines . . . . .	60
2.21	Sample optimistic execution . . . . .	63
2.22	Space of MPI checkpointing solutions . . . . .	85
2.23	Architecture diagram of RROMP . . . . .	91
2.24	Restart line made up of checkpointed process states and crossed by past, future, late and early messages . . . . .	93
2.25	Possible types of messages that may happen in protocol . . . . .	95
2.26	Checkpointing protocol pseudo-code . . . . .	101
2.27	Representation of blocking and non-blocking communication in abstract model . . . . .	105
2.28	MPI_Bcast interacting with checkpointing protocol . . . . .	110
2.29	Data flows of single-sender collectives . . . . .	111
2.30	Data flows of single-receiver collectives . . . . .	112
2.31	Data flows of all-to-all collectives . . . . .	112
2.32	Possible types of data flows that may happen in all-to-all collective communications . . . . .	114
2.33	Protocol for an all-to-all communication . . . . .	116
2.34	Protocol for single-receiver collective communication . . . . .	119
2.35	Protocol for single-sender collective communication . . . . .	123

2.36	Barrier crossing the restart line . . . . .	124
2.37	Protocol for barrier communication . . . . .	125
2.38	Example of a construction tree for datatypes . . . . .	129
2.39	Datatype creation . . . . .	130
2.40	Datatype piggybacking overheads for SMG2000 . . . . .	153
2.41	Benchmark diagrams . . . . .	154
2.42	Expected piggybacking bandwidth overheads . . . . .	156
2.43	Piggybacking bandwidth overheads (intra-node, unidirectional) . . . . .	158
2.44	Piggybacking bandwidth overheads (intra-node, bidirectional) . . . . .	159
2.45	Piggybacking latency overheads (intra-node) . . . . .	160
2.46	Piggybacking latency overheads (intra-node) . . . . .	161
2.47	<code>MPI_Bcast</code> . . . . .	163
2.48	<code>MPI_Gather</code> . . . . .	163
2.49	<code>MPI_Allgather</code> . . . . .	163
2.50	<code>MPI_Barrier</code> . . . . .	164
2.51	Application piggybacking overhead on 4 processors, Velocity 1 . . . . .	165
2.52	Application piggybacking overhead on 16 processors, Velocity 1 . . . . .	166
2.53	Piggybacking overhead for CG-C on 4-512 processors, Lemieux . . . . .	166
2.54	Piggybacking overhead for LU-C on 4-512 processors, Lemieux . . . . .	166
2.55	Piggybacking overhead for HPL on 4-512 processors, Lemieux . . . . .	167
3.1	Space of shared memory rollback restart solutions . . . . .	175
3.2	Overview checkpointing approach . . . . .	190
3.3	Blocking Checkpointing Protocol . . . . .	191
3.4	Deadlock example . . . . .	192
3.5	Checkpointing Protocol Framework . . . . .	196
3.6	Locks example . . . . .	200
3.7	Overview of the $C^3$ OpenMP Checkpointer . . . . .	209
3.8	Single Construct Example . . . . .	219
3.9	For Construct Example . . . . .	220
3.10	Restarting a simple <code>for</code> construct . . . . .	220
3.11	Multiple For Construct Example . . . . .	221
3.12	Checkpointing Protocol Extended with Potential Checkpoint Locations . . . . .	222
3.13	Critical example . . . . .	225
3.14	Sample application call stack . . . . .	245
3.15	Sample execution with <code>no_wait</code> parallel loops . . . . .	294
3.16	Sample execution of parallel <code>for</code> loop with <code>ordered</code> . . . . .	304
3.17	Example execution of <code>ordered</code> checkpointing algorithm. . . . .	305

# Chapter 1

## Introduction

High Performance Computing (HPC) is a branch of Computer Science that originated with the ENIAC and has flourished in various forms ever since. HPC systems have always included the fastest, largest and most capable computers in the world and the problem of creating them and using them to solve our largest computational problems has consistently been a special challenge, positioned at one of the extreme edges of Computer Science research. Over the past several decades, HPC systems have experienced a number of trends. Chief among them has been their increasing commoditization.

Early HPC systems like the ENIAC or the Cray-1 were specialized machines, custom-built for HPC work. However, advances in circuit design resulted in computers becoming increasingly cheaper, which led to a large market for much smaller computers. Based on microprocessors, these machines were much cheaper than their HPC counterparts but still quite capable for personal and server computing tasks. Over time, the much larger market for microprocessor-based computers resulted in tremendous amounts of R&D money being poured into research on better and faster microprocessor designs, dwarfing the money available in the much smaller HPC market. Over the past few decades HPC systems built out of commodity microprocessors, connected by high-quality networks have become dominant in the field.

Harnessing microprocessors to solve large computational problems has required the use of many such microprocessors in a single system. Whereas today the large server machines in the business sector may have as many as 32 processors, large

supercomputers can have thousands or tens of thousands of processors in a single machine. While this approach has proven itself to be highly effective in expanding the limits of computational capability, it has also brought to the foreground new challenges that did not arise in smaller systems. Fault tolerance is one such critical challenge.

The problem of fault tolerance in modern systems arises from two important HPC trends. First is the rising frequency of faults in systems. Second is the increasing size and running times of applications running on these systems, making them more vulnerable to these faults. HPC systems are vulnerable to faults for three major reasons. First, whereas older machines were built from custom-made, high-quality components, modern systems use commodity components that were designed and built for a less reliability-aware market. Second, as modern systems are made from more and more components, the probability of one of them failing becomes quite large, even if the individual components are reliable. Finally, as circuit feature sizes become smaller, circuits become increasingly vulnerable to soft, hard and intermittent errors, caused by ambient radiation, temperature fluctuation and other everyday phenomena. The end result is that the largest HPC systems today, such as the ASC systems, have mean times between faults on the order of hours to days. At the same time, many applications being run on these systems have running times on the order of days, weeks and months; for example, ab initio protein folding codes are projected to take on the order of years on BlueGene/L, the largest high-performance computer in the world.

The bottom line is that as HPC systems have grown in power, they have become less reliable. At the same time, many modern applications have placed ever-greater reliability demands on these systems. Since the design of HPC hardware is being

driven by external market forces, it is upto the software to become more tolerant of hardware faults. While it is possible to design applications to be insensitive to system faults [80] [114], this requires a significant amount of programmer effort and can be extremely difficult for general algorithms. A more general and simpler solution is *rollback restart*, a technique in which the state of the application is periodically saved to stable storage. In the event of a system fault the application is aborted and restarted on functioning hardware as if nothing happened. While less efficient than application-specific approaches, rollback restart has become popular because its simplicity leads to a lower time-to-solution than competing fault tolerance techniques.

In addition to being useful for fault tolerance, rollback restart is in fact a generic technique with a variety of applications. For example job scheduling becomes significantly easier if applications can be preemptively halted and then restored at a later time when more computational resources are available. Another example is debugging, where rollback restart can be used to execute the application in "reverse" by frequently recording its state, rolling the application backward and then executing it for a short time forward to simulate the reverse execution effect. Other examples include job migration, job monitoring, and guided simulation space exploration.

## **1.1 Rollback Restart**

### **1.1.1 Basic Checkpointing Techniques**

The overall problem of rollback restart can be described in a variety of different ways depending on whether it is performed in the context of sequential vs parallel

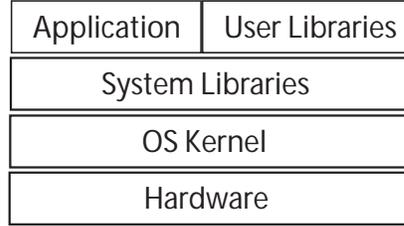


Figure 1.1: Sequential system stack

applications, interactive vs non-interactive applications, etc. For this introductory discussion we will consider the problem of rollback restart as the problem of checkpointing and restarting a sequential application (single process, single thread), with related problems of parallelism and external interactions modelled as saving the state of external libraries that the application may be linked with.

The sequential checkpointing problem is easy to state: the state of the application must be saved and later restored on the same or different system. However, there are a large number of design choices. To understand this consider the typical system stack shown in Figure 1.1. It contains the original user application, which may be compiled/linked with user-level libraries. It may use system libraries, which reside on top of the OS kernel, which executes directly on top of hardware. Any of these levels may be modified with checkpointing functionality, which makes it possible to save and restore the stack levels above it. Furthermore, it is possible to insert new layers between these standard layers that enable checkpointing of the layers above.

For example, consider hardware-level checkpointing. It is possible to modify system hardware [175] [103] to enable it to record its own state. This makes it possible to save the state of any software working on top of this hardware. Another option is to use virtual machine technology [64] to insert a layer between the OS kernel and the hardware, checkpointing the software stack on top on existing non-

checkpoint-enabled hardware. Going up the stack, it is possible to create kernel-level [75] and user-level [169] solutions that sit at their respective levels in the system stack below the application. Finally, it is possible to modify the application itself, either manually [203] or automatically [178], to enable it to checkpoint its own state.

### 1.1.2 The Four R's of Checkpointing

While it is possible to solve the problem at any level, different levels present different design challenges. Placing checkpointing functionality at a given level is like creating a virtual machine that pretends to the stack levels above it to be identical to the stack levels below it, except that it is able to maintain this illusion even if the levels above are shut down and restarted somewhere else. Thus, the checkpointing solution must use the API provided by the levels below to simulate lower level functionality while adding additional restart functionality. For example, consider a kernel-level checkpointer. It has full access to the state of all applications as well as the operating system. However, while it can send requests to different hardware devices, it does not have direct access to their state. This includes the CPU itself, which contains a great deal of internal state (pipeline, predictors, speculative state, etc.) that is completely invisible to the software. Thus, while a kernel-level checkpointer can freely save application and OS memory state, it needs to use available hardware APIs to record and save software-visible hardware state. This would include the contents of registers but would not include things not visible to software like predictor and speculative state.

Speaking more generally, depending on the kinds of access that a checkpointing solution at a given level has to lower-level state, checkpointers have as many as

four mechanisms at their disposal to record and restore application/system state.

They are: (the examples provided are for a kernel-level checkpointer)

- **Restore:** State that can be directly manipulated by the checkpointer and can therefore be directly saved at checkpoint-time and restored on restart. Example: application and OS state.
- **Replay:** State that cannot be directly manipulated by the application but can be recreated using a sequence of deterministic operations. On restart it can be regenerated by replaying these operations. Example: resending messages interrupted by the checkpoint to their respective destinations.
- **Reimplement:** State that cannot be recreated by replaying operations. In this case the operations that create and maintain this state need to be reimplemented so that this state can be saved and restored. Example: an FPGA that contains a stateful circuit and does not provide an API for external access to this state.
- **Restrict:** State that cannot be recreated by reimplementing the operations. This type of state cannot be supported and the application must be restricted from using this kind of state or to only using it in a restricted manner. Example: specialized security hardware that contains state about the current session that cannot be read or restored (by design) and the functionality of which cannot be replicated because it depends on a secret key within the hardware.

This taxonomy can be applied to solutions at any level and offers a simple and uniform way to talk about the implementation complexity tradeoffs inherent

in creating checkpointing solutions at different levels of the system stack. Another example might be an application-level checkpointer that modifies application source code to make it self-checkpointing and self-restarting. While such a checkpointer would have direct access to its own state, it would have no such access to any libraries linked into the application, the internals of the OS or of the hardware. In particular, it would have to use Replay to recreate the structure of the call stack, calling the same sequence of functions on restart as were invoked on the stack at the time of the checkpoint. Furthermore, while `malloc` allows applications to allocate memory on the heap, it provides no API for choosing where the allocated heap buffers will be placed. As such, an application-level checkpointer must use Reimplement to create a new self-checkpointing implementation of `malloc`. Such an reimplement would be based on functions like `mmap` on Unix or `VirtualAlloc` on Windows that provide an API that is flexible enough to be used with Replay. Once the call stack structure and heap buffer locations have been restored, Restore can be used to reset the values of stack variables and the heap buffers themselves to their original values. The state of the OS and the hardware would be recreated using a variety of techniques that depend on the state in question. For example, file I/O could be dealt with via Replay since such APIs (Unix and Windows file I/O in particular) make their key internal state (status of files, file pointers, etc.) visible to the application.

### 1.1.3 Current State of the Art

While there has been significant research on checkpointing techniques, spanning over three decades, this work has unfortunately had little practical impact. Most HPC system manufacturers do not provide checkpointing with their machines.

Furthermore, a brief survey of the major US supercomputing centers and government laboratories shows that most clusters do not have automated checkpointing available on them (the exceptions are for the most part IBM machines and Cray machines before the XT3) and even for those that do, their use is generally discouraged by system administrators. There are two major reasons for this. First, HPC applications tend to run on large numbers of processors and produce large amounts of state. As such, the basic cost of shipping checkpoint data to disk becomes so large as to become impractical. Second, typical approaches to checkpointing are implemented at such a low level that they are non-portable, leaving most real applications and machines without a viable checkpointing solution.

The first problem is as much reality as it is myth. While it is true that large applications require significant amounts of memory, this does not need to translate into expensive checkpoints. Incremental checkpointing [166] and compression [170] are two automated techniques that can significantly reduce pressure on the network and the I/O subsystem. Indeed, [120] reports that incremental checkpointing on its own can make checkpointing of common large applications practical on existing I/O and network hardware without additional optimizations. While this is a useful result, it is not universal. Some systems, such as the BlueGene/L, do not provide virtual memory, which is generally needed to implement incremental checkpointing. Furthermore, because incremental checkpointing extends the amount of time required to complete a single checkpoint, it is a poor choice for situations where fast checkpoint completion is important, such as job scheduling. The alternative is to reduce the amount of data being saved.

Both [170] and [133] experimentally evaluate checkpoint compression and while they both report high compression ratios on their benchmarks, [133] finds the cost

of compression to be too high to be practical while [170] finds it to be beneficial, leaving the ultimate usefulness of compression unclear. Systems like [203] and [197] allow the programmer to reduce the amount of state saved by manually identifying the regions of memory to be saved. [168] extends this idea with a compiler analysis that uses programmer annotations about what state to exclude from a checkpoint to determine what state does not need to be saved at specific locations in the source code. In light of this work, it should be clear that while large checkpoints are a real problem in HPC, there do exist viable solutions to this problem. Unfortunately these solutions are rarely available on real systems due to the problems discussed below.

The second impediment to broad acceptance of checkpointing technology is a general lack of solution portability. The vast majority of checkpointing work has focused on monolithic systems implemented either inside the OS kernel or immediately above the OS interface. While this approach has made it easier for a single group to develop something that works, solutions of this sort generally become locked into the platform for which they were originally designed.

The close ties checkpointers typically have to their host operating systems make it very difficult to extend their range of supported operating systems without significantly rewriting the code base. In particular, a Linux checkpointer works only on Linux and would require significant porting work to work on any other Unix, with little realistic possibility for porting to unrelated operating systems such as Windows. Similarly, the typical approach to checkpointing parallel communication APIs, such as MPI, has been to modify existing implementations of such APIs and encourage HPC users to adopt them on their systems. This is not a practical approach since communication libraries are highly tuned to the details of the systems

for which they are optimized so users have little motivation to drop their highly tuned libraries and adopt third party libraries that may be checkpoint-enabled but offer little in terms of performance. As such, typical checkpointers support their chosen operating systems and library implementations and can help users with little else.

The monolithic structure of existing checkpointing systems has further helped to limit the range of libraries and services that they support since it is difficult for other groups to add to them additional checkpointing capabilities. BLCR [75] is one exception in that it provides a limited API for checkpointing resources/libraries that are not checkpointable by the basic system. However, its facilities are limited to a few callbacks and even at this level of extensibility, it is already one of the most extensible checkpointing systems around. This lack of modularity has had a negative impact on the adoption of checkpointing technology because most checkpointers are limited in the libraries, system calls and hardware that can be supported by their original development group and as such, are applicable to relatively small numbers of real applications and real machines.

#### **1.1.4 Practical Checkpointer Design**

In designing a practical checkpointing solution it is imperative to trade off implementation complexity versus solution generality. In general, lower-level solutions are simpler because they have more direct access to more parts of the system while higher-level solutions have less access and must therefore use complex APIs to perform the necessary state saving and restoring. This general pattern does not always hold, however, since higher-level solutions can use higher-level semantics to avoid saving parts of system state, which can help to reduce their complexity. Another

factor is that lower-level solutions tend to be less general, while higher-level solutions more general. For example, a checkpointing given solution for a given type of hardware is rarely directly applicable to other types of hardware. Similarly, a checkpointer for one OS or kernel requires significant work to be ported to another OS or kernel. In contrast, application-level solutions are very general since they can modify applications to become self-checkpointing and self-restarting on any platform on which they may run. The net effect is that there is a balance to be found between implementation complexity and generality that minimizes the overall effort required to create a checkpointing system that can be used by the vast majority of applications on the vast majority real machines. Finding the appropriate balance, however, is generally difficult and must be done on an API-by-API basis if we hope to find the true optimum.

A given application is some combination of languages, libraries and hardware, with each language, library and hardware presenting its own complexity/generality trade-off. Thus, the optimal checkpointing solution would look at all possible APIs that may be used directly or indirectly by any application and for each API create a checkpointing solution at the level most appropriate for that API. This approach would make a given checkpointer a modular framework that can be instantiated with a heterogeneous family of API-specific sub-checkpointers, each implemented at the optimal level for their respective target APIs.

For example, consider an application written in C++ that uses a BLAS library (basic linear algebra routines), some implementation of MPI (a communication API) and an Oracle database. While [139] shows how a C application may be transformed by a pre-compiler to become self-checkpointing and self-restarting, C++ applications have much more complex state that is extremely difficult to

record using language-level techniques. As such, [139] also presents a user-level checkpointing system that works below the language level to save the state of the application, an option that is most appropriate in the case of C++.

BLAS libraries are high-performance implementations of linear algebra operations that are typically highly optimized for a given architecture but other than basic CPU instructions do not touch the hardware directly. Typical implementations of such libraries rest above the operating system and use little OS functionality besides the memory allocation routines. As such, their state can be checkpointed trivially by linking a checkpointing module with the BLAS library that records the library's memory state. If such a library makes non-trivial use of OS system calls, the relevant OS state can be checkpointed by sub-checkpointers dedicated to the APIs in question and implemented at their respective optimal level.

MPI is a high-performance communication API that is usually implemented by having the OS grant the MPI library direct access to the networking hardware. As such, it may appear that the most appropriate way to checkpoint MPI implementations is to do so at a low level, by modifying a specific implementation of MPI or by placing a layer between the MPI implementation and the network hardware (most prior work has taken this direction). However, Chapter 2 argues the opposite: that the most appropriate way to checkpoint MPI implementations is via a layer above MPI that is compatible with any implementation of this API. As Chapter 2 shows, this can be done efficiently on multiple operating systems and implementations of MPI.

Finally, databases such as Oracle contain very complex internal state and generally bypass most OS resource allocation mechanisms, preferring to allocate disk and RAM storage on their own. However, they also provide applications with sig-

nificant access to their stored data via SQL queries and stored procedures. At this time little research has been conducted on the appropriate way to checkpoint the state of databases. However, decades of research on transactional memory in such databases suggests that it should be possible to efficiently checkpoint database state both from inside the database implementation, as part of the transaction manager, as well as from above the database, placing checkpointing code into transactions.

## 1.2 Organization of Thesis

This thesis advances the state of the art in the development of modular checkpointing frameworks by addressing the problem of extending single-process and single-thread checkpointers to provide rollback restart for parallel applications. Chapter 2 focuses on rollback restart for message passing applications. It presents a novel checkpoint coordination protocol for such applications that is new in being compatible with sequential checkpointers that may not be able to take a checkpoint at any point during execution. It then applies this protocol to the problem of checkpointing applications written for the MPI message passing API and shows how to deal with various complex features of MPI including collective communication and non-blocking communication. Chapter 3 looks at rollback restart for shared memory applications. It presents a generic checkpointing protocol for shared memory applications running on top of arbitrary shared memory models, consistency protocols, etc. It then experimentally proves the generality of this protocol by applying it to the OpenMP shared memory API and showing that the resulting checkpointing solution has low overhead. Future extensions of the work presented in this thesis are discussed in Chapter 4.

# Chapter 2

## Checkpointing Message Passing

### Applications

#### 2.1 Background

Message passing is a way for different threads of execution within a single application to communicate with each other. In this chapter these threads are called "tasks", although in practice they may be implemented as processes, threads, or using some other mechanism. The message passing programming model consists of two things: distributed memory and sender-receiver communication. Distributed memory means that the address space available to the application is disjoint and each task can only access its own portion of the address space. Communication between tasks is enabled by `send` and `receive` operations where a given `send` matches a single `receive` and vice versa. Popular APIs for message passing include MPI [1], TCP [22], UDP [174] and PVM [92]. Over the past few decades there has been a great deal of work on providing rollback restart for parallel message passing applications, dealing almost exclusively with blocking point-to-point sends and receives (the basic primitives common to all message passing APIs). This section surveys this prior work.

In the context of message passing applications, rollback restart means that one or more tasks roll back to a prior state and continue their respective executions. The job of a rollback restart system is to ensure that this restart process does not violate application semantics (i.e. from the perspective of an external observer

the application that restarted should behave just like an application that did not). In the absence of communication between the application's tasks, this reduces to the problem of sequential rollback-restart, once for every task that rolled back. The novel thing in message-passing applications is the set of inter-dependencies between the application's tasks induced by their message-based communication.

In order to provide the application with the illusion of no-rollback it is necessary to be able to monitor and control the application's interactions with the message passing system. As such, existing work on this problem assumes that in addition to a sequential checkpointing system that can roll back individual tasks, there exists a layer of code between the application and the message passing system, shown in Figure 2.1. This rollback restart layer virtualizes the message passing system, providing to the application functionality that is identical to what is available from the underlying communication system, plus the ability to restart one or more application tasks. The rollback layer is able to call functions from the underlying communication system and each time the application calls a communication operation, it is able to execute arbitrary code. In most protocols the rollback layer passes the request on to the underlying communication system and performs some logging operations required to perform a successful restart. In all cases the layer must ensure that:

- the application cannot distinguish between the original communication system and the communication system with the rollback restart layer running on top of it and
- the application cannot distinguish between the original communication system + rollback restart layer in the presence of task rollbacks and the original communication system with no task rollbacks.

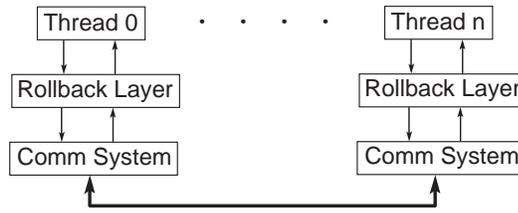


Figure 2.1: Rollback restart layer between the application and the communication system

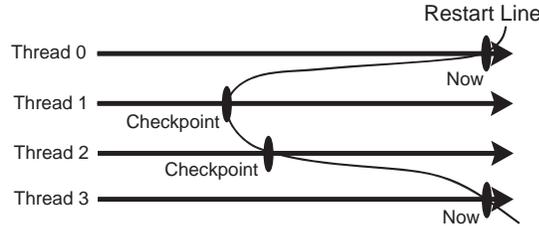


Figure 2.2: Example restart line

To understand the issues of providing rollback restart for parallel message passing applications we need a way to represent the state of such applications after one or more tasks have rolled back to prior checkpoints. This global configuration is captured via the notion of the "restart line", an imaginary line connecting the execution states of different tasks, one state per task. Each task's state identifies the configuration from which it will resume its execution. For tasks that were rolled back, this is a prior state retrieved from a checkpoint. For the remaining tasks this is their current state. Figure 2.2 shows a sample restart line. The horizontal lines are the time lines for different tasks and the ovals are the states from which each task will continue its execution (they represent some points in time on each task). The line connecting the ovals is the restart line and in this example it consisting of the present states of tasks 0 and 3 and the rolled back states of tasks 1 and 2.

The restart line is used to analyze what may happen if every task begins executing from its respective state. The general notation used to describe restart lines and the checkpoints they contain is used in Figure 2.3. Each local checkpoint is

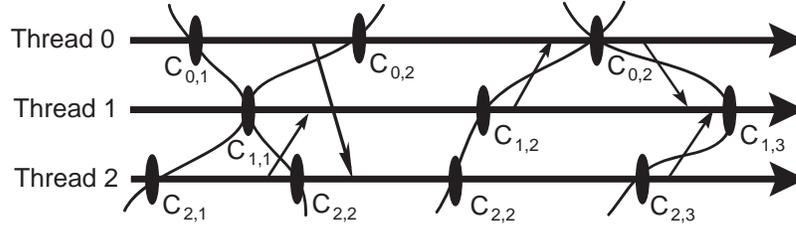


Figure 2.3: Example of checkpoints and restart lines

denoted via  $C_{t,c}$ , where  $t$  is the id of the task that took the checkpoint and  $c$  is the index number of the checkpoint ( $c$  starts at 1 for each task, with  $c = 0$  representing the start of the task's execution and  $c = \infty$  representing the task's current state). A restart line of an application with  $n$  tasks is denoted by  $(C_{0,d_1}, \dots, C_{n,d_n})$ , with  $d_t$  being the index number of some local checkpoint taken by task  $t$ . The time between two checkpoints on a given task is called an "inter-checkpoint interval" or simply "interval", with the interval between checkpoints  $C_{t,d}$  and  $C_{t,d+1}$  being denoted by  $I_{t,d}$ .

Events that happen on a task after it has taken a local checkpoint are called "beyond" it while events that happened before the local checkpoint are called "behind" it. Extending this terminology, an event is "beyond"/"behind" restart line if it is beyond/behind the local checkpoint on its task that is part of this restart line.

This notation is used to analyze the types of communication that may exist relative to each line. For every type of communication we must examine how it is seen by each task and how this impacts our ultimate goal: providing the application with the illusion that no task has been rolled back to a prior state.

In total, there exist four types of messages that may exist relative to a restart line, each of which is shown in Figure 2.4. The straight line with arrows represent point to point messages. The tail of an arrow represents a call to `send` while its

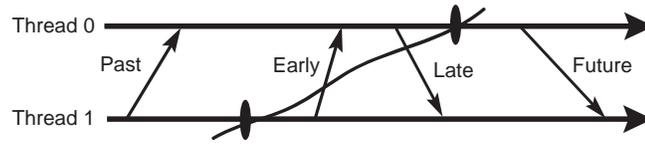


Figure 2.4: Restart line with Past, Future, Late and Early messages

head represents the matching call to `receive`.

We can now examine each type of message and analyze what will happen if each task continues its execution from its respective state.

- **Past Messages:** These are messages sent and received before their respective tasks' restart states. When each task continues executing, each task will believe that these messages have been sent and received and has the same opinion about the data of these messages. Thus, there is no inconsistency in the application and the illusion that no task rolled back is maintained.
- **Future Messages:** These messages were sent and received after their respective tasks' restart states. As tasks continue their execution each task will believe that it has not yet sent or received any future message. As such, their states are consistent and as they execute, they may or may not re-send/re-receive these messages, as dictated by application semantics (deviation from prior executions may occur due to non-deterministic operations). Again, the illusion of no rollback is maintained.
- **Late Messages:** A late message is one that is sent *before* its sender task took its checkpoint and received *after* its receiver task took its checkpoint. On restart, the sender task will believe that it has already sent the late message (in fact, by that point in time it may have already overwritten its data) while the receiver task will believe that it has not yet received this message. Thus,

as tasks continue their executions, the sender task will continue executing without re-sending the late messages. Meanwhile, as the receiver continues its execution, it may eventually re-execute its `receive` operation, expecting to get its data. (“may” because we do not assume that all sent messages must be received)

To deal with any `receive` of any such message, a rollback-restart system must record the data of any late messages or just sufficient information to recompute this data on restart. When receiver tasks re-execute their `receive` operations for the late messages, the recorded/recomputed message data needs to be fed back to them. This can be done by either reissuing matching sends with the original message data or feeding the message data directly to the application’s `receive` operation without actually posting a receive on the network. Note that in the presence of non-determinism it is possible for the receiver thread to execute a different sequence of operations on restart. As such, the `receive` operation that receives a given late message on restart may not be the same one that received it during the original execution.

In prior literature, Late messages are also known as “in-flight messages” because if one considers the restart line to be a possible point in time in the parallel application’s execution, these messages would have been in-flight on the network at that time.

- **Early Messages:** An early message is one that is sent *after* its sender task took its checkpoint and received *before* its receiver task took its checkpoint. In their respective restart line states, the sender task will believe that it has not yet sent the early message while the receiver task will believe that it

has already received it. Because it is not possible for one task to receive a message before another task has sent it, the existence of an early message implies that the sender task was rolled back from a state after it sent the early message to a state before the send.

When the receiver task continues executing from its restart line state, it will not re-execute the `receive` operation. Meanwhile, as the sender task resumes its execution, two things may happen. If during the sender task's pre-restart execution there were no non-deterministic events between the checkpoint and the early message's send operation, then the sender will re-execute the `send` operation with its original data. Since the receiver task will not re-execute the `receive` operation, this creates an inconsistency between the two tasks. This can be resolved by making sure that the sender's re-executed `send` operation is not performed on the network or by forcing the receiver task to execute a matching `receive` operation.

If there were any non-deterministic events between the sender's checkpoint and the send of the early message then on restart the sender may not resend the message or may resend it with different data. Since on restart the receiver task believes that it has received this message and that this message had its original data, if the sender does not send the message with that same data, another inconsistency is created between the two tasks. Since this inconsistency is caused by non-deterministic events having a different outcome on restart, the way to avoid it is to

- save the outcomes of all non-deterministic events that happen between the sender's checkpoint and the `send` operation of the early message;

- on restart force all such events to occur exactly as they were recorded to.

If this is done then we can be sure that the sender task will re-execute the `send` operation with the same data as on restart, meaning that the non-deterministic case reduces to the deterministic case above.

## 2.2 Taxonomy of Solutions

Having defined restart lines, their constituent task states and the types of messages that may exist relative to them, it is now possible to provide a useful taxonomy of rollback restart protocols based on the type of restart line they generate. This is an incremental taxonomy, based on two orthogonal classification schemes.

### Restart States:

The first classification scheme is a hierarchy that relates the sets checkpointing and message logging protocols.

- **Checkpointing** protocols generate restart lines where all tasks have rolled back to a prior checkpoint.
- **Message Logging** protocols generate restart lines where some tasks may not have rolled back and their restart line state is in fact their current state.

Since the message logging allows for all processes to rollback, it is a larger class of protocols than checkpointing. However, the fact that checkpointing is a more constrained problem allows for a much greater freedom of solutions and is thus examined separately from message logging.

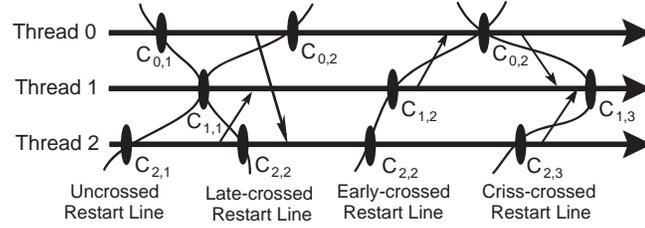


Figure 2.5: Example of checkpoints and labeled restart lines

### Types of Restart Lines:

The second classification scheme divides restart lines into equivalence classes according to the types of messages that may cross them. Between our four possibilities, (past, future, late and early), there exist four interesting types of restart lines:

- **Uncrossed** restart lines are not crossed by any messages, meaning that only past and future messages may exist relative to such restart lines.
- **Late-Crossed** restart lines are crossed only by late messages.
- **Early-Crossed** restart lines are crossed only by early messages.
- **Cris-Crossed** restart lines are crossed only by both late and early messages.

Figure 2.5 provides an example by taking Figure 2.3 and labeling each restart line with its type.

The primary value of this classification is twofold. First, by identifying the types of messages that may cross a restart line, it clearly identifies the invariants that must be maintained by the rollback restart protocol. Second, the types of messages that cross a restart line of a given type place constraints on the situations when such restart lines may be useful.

## 2.2.1 Applying the Taxonomy

The above classification schemes create a taxonomy of possible rollback-restart solutions that contains 8 types of restart lines. While some possibilities do not contain any protocols, others are rich in structure and in this section we look at the possible classifications to understand their properties and their applicability to real rollback-restart situations.

### 2.2.1.1 Checkpointing

The first classification to examine is the Checkpointing class of restart lines, where all task states that make up a restart line come from a checkpoint of a prior task state, rather than some task's current state. Since there are few constraints on the placement of such checkpoints relative to communication events, such restart lines may or may not be crossed by late or early messages. This means that checkpointing restart lines may be Uncrossed, Late-crossed, Early-crossed and Criss-crossed.

Uncrossed restart lines are the most constrained type of line and are thus the most difficult to create. Since many applications have outstanding communication at all points in their executions, it may not always be possible to create such lines. The code fragment in Figure 2.6 is an example of this. If it executed by two tasks, the message sent by task 0 will be in-flight for almost all of the application's execution. Furthermore, as discussed in Section 2.3.1, the generation of such restart lines is complex and may cause tasks to roll back a significant amount of their computation. Despite the complexity of creating an Uncrossed restart line, it presents some useful benefits. The foremost is the simplicity of reasoning about such lines. Since an Uncrossed restart line corresponds to an actual state of the application at some point in its execution, its correctness is clear.

Furthermore, the simple semantics of Uncrossed restart lines make them much more human-understandable, which can be useful in debugging and state visualization. Another positive aspect is the potentially reduced amount of storage needed to record such restart lines due to the fact that no information about messages or non-determinism needs to be recorded, as it does for more complex restart lines.

```

if(taskID==0) {
    send(data, task1);
    ... body of application ...
} else if(taskID==1) {
    ... body of application ...
    recv(buf, task0);
}

```

Figure 2.6: Code example where all restart lines are crossed by a late message

Late-crossed restart lines are less constrained than Uncrossed lines but still quite useful. Because of their fewer constraints they can be found on any execution of any application. The literature on protocols to find them is varied and rich. Although no useful Uncrossed restart lines can be formed in the example in Figure 2.6, any local checkpoints taken on the two tasks can be combined to form a Late-crossed restart line. More generally, as long as each task is free to checkpoint at any time during its execution, it is possible to find a Late-crossed restart line in any execution.

For an intuition of why this is true, examine Figure 2.7. On the left is an example restart line  $(C_{0,i}, C_{1,j}, C_{2,k})$ , which is crossed by early messages. Although this restart line is not Late-crossed, it can be made so by forcing tasks 0 and 2 to record local checkpoints immediately before they receive the early messages. This

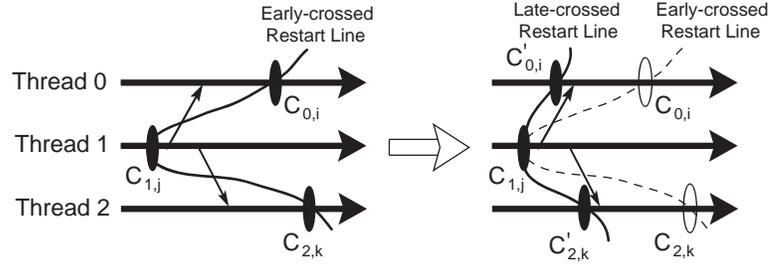


Figure 2.7: Example of late-crossed restart line creation

results in restart line  $(C'_{0,i}, C'_{1,j}, C'_{2,k})$  on the right, which is Late-crossed. Thus, given the ability to take a checkpoint at any time during a task's execution, any non-Late-crossed restart line can be made Late-crossed by taking every early message that crosses the restart line and forcing each receiver task to take a checkpoint immediately before it receives any such message. It can be shown that it is always possible to find a location to force a checkpoint without causing new messages to be early. (the proof is by contradiction: if it is not possible then the messages in question form a causal cycle, which is impossible)

Despite the fact that it is always possible to find Late-crossed restart lines Late-crossed lines are not guaranteed to correspond to a state of the application at some point in its execution (as is the case for Uncrossed restart Lines). However, it can be shown [59] they correspond to some legal state of the application, which may or may not have happened during the given execution. In general, this does not make it more difficult to reason about Late-crossed restart lines than Uncrossed ones but it reduce their usefulness for debugging purposes since such restart lines do not represent what actually happened during the execution.

Early-crossed restart lines are interesting because they are intuitive and easy to define while being mostly useless. The reason is that while Uncrossed and Late-crossed restart lines correspond to an actual or possible state of the application, Early-crossed lines do not. The reason is simple: an early message is one

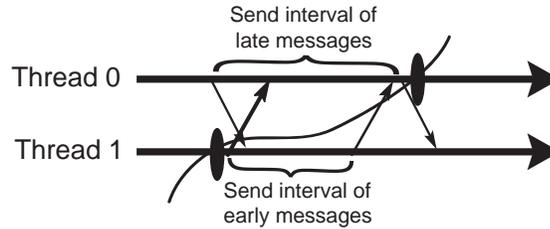


Figure 2.8: Time intervals of when early messages may be sent vs when late messages may be sent.

that is sent after its sender task's restart state and received before the receiver task's restart state. As such, the restart line corresponds to the impossible state where the messages has been received but not yet sent. In addition to being hard to reason about, Early-crossed restart lines are quite rare. Figure 2.8 shows this phenomenon. For a given restart line and a given average message latency (indicated by the slope of the message lines), the amount of time during which sent messages may become late relative to the restart line is slightly longer than the corresponding time interval for early messages. This means that any restart line that is crossed by an early message is likely to also be crossed by a late message, making it a Criss-crossed than an Early-crossed restart line.

While it is possible to create protocols that would specifically eliminate late messages while allowing early messages, such protocols would have dubious value. The cost of late messages is the overhead from storing their data (or enough information to reconstruct their data). The cost of early messages is the storage cost of saving their IDs and the cost is recording all non-deterministic events that precede the sending of any early message. Such non-deterministic logging can be complex to implement (or impossible in certain situations) and may generate a great deal of data that must also be stored in reliable memory (e.g. saving all the non-deterministic decisions of a shared memory library). Since the relative use-

lessness of Early-crossed restart lines, does not warrant the cost of creating them, there has been no work on protocols that can generate them.

Criss-crossed restart lines are much more useful than their Early-crossed counterparts. While prior work on the generation of Late-crossed restart lines assumes the ability to force a checkpoint on any task at any point in time, in practice this capability is not always available or desirable. Most checkpointing experiments have found that the dominant cost of checkpointing is the time to write the states of the tasks out to disk rather than any cost associated with generating restart lines. This overhead can for the most part be attributed to two sources:

1. the pressure placed on the reliable checkpoint storage system by having a large number of tasks writing their state at the same time, and
2. the size of each task's checkpoint.

The pressure on the storage system resulting from many tasks simultaneously recording checkpoints can be resolved by spacing such checkpoints out in time. One way of doing this is via incremental checkpointing [166], which requires support for virtual memory and the availability of local unreliable memory (such as unused RAM or local disk) for use as write buffers, either of which may not be available. A much simpler option is to simply space the task's local checkpoints out in time. However, the larger the time gap between tasks' checkpoints, the larger the probability of messages crossing the resulting restart line, causing it to become Criss-crossed.

The size of the checkpoints can be reduced via general techniques such as compression [170] but in practice using application semantics leads to more significant reductions in checkpoint sizes, via a technique known as Application-level Check-

pointing. This can include both asking the programmer to identify the parts of applications state to save and/or not to save [203] as well as compiler analyses that do the same automatically [168]. In both cases, it is necessary to identify specific points of the application (called "potential checkpoint locations" or "PCLs") that have unusually little state to save and restrict the application from checkpointing anywhere else in its code.

The problem with limiting the times when tasks can record local checkpoints is that it naturally causes the resulting checkpoints to form Criss-crossed restart lines. Consider the example code in Figure 2.9, sample execution of which is shown in Figure 2.10. The way the messages and the PCLs get ordered, every possible restart line is crossed by an early message. (we can make sure that they are also crossed by late messages the same way as in the example in Figure 2.6) If we had full freedom of when to have each task take a checkpoint, it would be possible to avoid the early messages by forcing each task to checkpoint immediately before the early message receives. However, given the constraint that checkpoints may only be taken at PCLs, this is not possible, which makes Criss-crossed restart lines inevitable in this case.

It should be noted that while Criss-crossed restart lines allow for greater efficiency in taking and recording local checkpoints, the fact that they may be crossed by both late and early messages cause overheads and complexities due to the logging of message data and identifiers as well as non-deterministic events. While these need to be saved only for messages and non-deterministic events that happen at the time of the checkpoint, in some contexts they may become a dominant overhead, undoing the advantages that Criss-crossed restart lines have in state-saving overhead.

```

while(1) {
    if(taskID==0) {
        recv(buf, task1);
        potential_checkpoint();
        send(data, task1);
    }
    else if(taskID==1) {
        potential_checkpoint();
        send(data, task1);
        recv(buf, task1);
    }
}

```

Figure 2.9: Code example where all restart lines are crossed by early messages

Out of our four possible types of restart lines, all four may happen in the context of checkpointing. While Early-crossed lines turn out to be mostly useless, Uncrossed, Late-crossed and Criss-crossed restart lines all have their uses, depending on the various sources of overhead in various situations. Section 2.3 examines these issues more closely, providing more detail about the many varieties of checkpointing protocols found in literature.

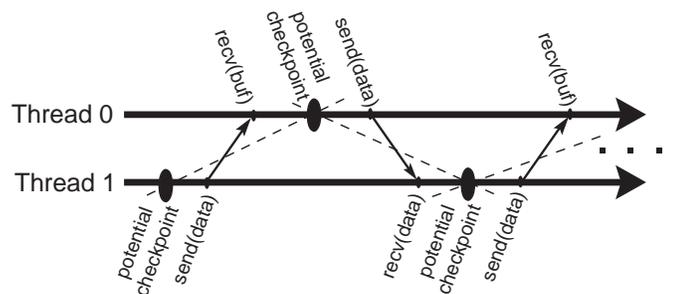


Figure 2.10: All restart lines crossed by early message

### 2.2.1.2 Message Logging

Having examined the types of restart lines that may happen in the context of checkpointing, we can now look at what happens in message logging. Recall that a message logging restart line is one where not all tasks have rolled back, meaning that there may be one task whose restart state is its current state (it did not roll back) and there is at least one task whose restart state is from a prior checkpoint (meaning that a rollback has occurred).

To see what this implies, consider Figure 2.11. The restart line in this example contains the current states of tasks 0 and 1 and checkpoint states of tasks 2 and 3. The portion of the restart line between tasks 0 and 1 is simple. Since it is possible for messages that were sent before task 1's current state to arrive after task 0's current state, it can be Late-crossed. However, it cannot be crossed by early messages since there can be no messages sent after either task's current state. As such, this portion of the restart line may be Uncrossed or Late-crossed. The portion of the restart line between tasks 2 and 3 is more complex in that it may be Uncrossed, Late-crossed, Early-crossed or Criss-crossed, depending on how or whether their checkpoints were coordinated, as discussed above. Finally, consider the portion of the restart line connecting tasks 0 and 1 (current states) and tasks 2 and 3 (checkpoint states). Since this is a parallel application, its tasks will periodically send and receive messages. However, in the time period after the checkpoints on tasks 2 and 3, any messages sent to them from tasks 0 and 1 will automatically be late. Similarly, any messages sent in the other direction will automatically be early. This means that unless a checkpoint is taken immediately after every send and/or receive (something that is very expensive), this portion of the restart line will necessarily be Criss-crossed since it is not known a priori

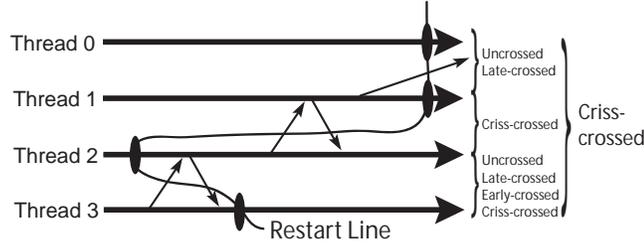


Figure 2.11: Example message logging restart line

which tasks will roll back.

Thus we see that any (efficiently generated) message logging restart line will have a portion that is crossed by both early and late messages. This makes the entire restart line Criss-crossed, requiring message logging protocols must record the data of all late messages, ids of early messages and all non-deterministic events that may ever happen (since any such event may be followed by the send of an early message). While in theory, by making assumptions about the communication pattern of the application and placing limits on which tasks may roll back (e.g. task  $t$  may only roll back if task  $r$  does as well), it may possible to force all real restart lines to be Late-crossed or Uncrossed, this possibility has not been explored and thus far there has only been work on message logging protocols with Criss-crossed restart lines. This work is described in Section 2.4.

## 2.3 Checkpointing Protocols

The past three decades have seen the development of many checkpointing protocols [79]. While some focus on distributed systems with relatively little communication, others target the HPC environment, with its high communication and tight latency requirements. Some protocols rely on application information while others treat the application as a black box. Finally, the variety of message passing models

has spawned various protocols that target their particular semantics. As such, in additions to protocols that target basic blocking point-to-point sends and receives there exist protocols that support unreliable message passing and protocols that support more complex message passing primitives such as broadcasts and reductions. The following sections will discuss the major categories of checkpointing protocols. Each category will be associated with its respective type of restart line.

### 2.3.1 Communication-free protocols

This is the class of protocol that generate **Uncrossed restart lines**. They come in two varieties: those that rely on application semantics to avoid late and early messages and those that do not. Because there is no communication happening across the restart line, such systems can use the trivial checkpointing protocol where the message passing system is effectively ignored and the state of each process is checkpointed independently using sequential checkpointing techniques. On restart each process can be restarted from its individual checkpoint and resume execution, without doing anything special to the underlying communication system (except for reinitializing it).

If the message-passing system does not guarantee the reliable delivery of messages (e.g. IP or UDP), finding Uncrossed restart lines is not that hard. Suppose that we have a set of checkpoints that form a Late-crossed restart line. Since there is no guarantee that any of the late messages that crossed this line will actually be delivered to the recipient, it is legal to treat these messages as lost and treat the restart line itself as Uncrossed. As such, any protocol that can generate Late-crossed restart lines can be transformed into an Uncrossed restart line protocol if it is used with an unreliable message-passing system. [110] uses this insight to

develop an efficient message logging protocol for such communication systems.

### 2.3.1.1 Application-Aware Protocols

Protocols in this class rely on the application programmer to tell the checkpointing system when it can checkpoint the state of each process so as to make sure that no message may cross the resulting restart line. One checkpointing system to make such a requirement is the checkpointer employed on BlueGene/L [27] [145], which requires that "When a process makes a call to BGLCheckpoint() there are no outstanding messages in the network or buffers (i.e. the recv corresponding to all the send calls have taken place)." [145] ([69] makes the same assumption) While it may be difficult to ensure invariant in general applications, this becomes easier for applications written in more constrained programming styles.

One popular and very simple style of parallel applications is known "Embarrassingly Parallel". Such applications can be broken up into a set of independent tasks that can be run on different pieces of hardware with no communication between them. The prototypical example of such an application is SETI@Home [127], where independent data processing tasks are simultaneously performed by millions of computers. In the absence of inter-task communication, it is trivial to create a restart line that connects checkpoints on all tasks and is not crossed by any messages. Other examples include Monte Carlo [86] simulations and rendering movie frames in parallel.

While in embarrassingly parallel applications there is little to no interaction between the individual tasks, the master-slave programming model [186] extends the embarrassingly parallel model by adding a "master task". This master task can be thought of as a sequential process that can call subroutines that execute on

other processors. These subroutines, called "slave tasks", do not communicate with each other and only communicate back to the master when they have computed their result. Since master-slave computations do feature communication, it is not trivial to place checkpoints on all tasks such that they make up a restart line that is not crossed by messages. However, master-slave is special because in most real applications the slave tasks do not run for a long time. As such, their initial state is a perfectly adequate checkpoint. Thus, for such applications it is easy to construct a restart line that is not crossed by any messages by using any checkpoint of the master task and the job descriptions of all the slave tasks that were executing at the time the master's checkpoint was taken. The application's state can be restored by restoring the master task to its prior state and restarting the slave tasks from the beginning.

The above styles of parallel programming severely limit the types of communication that may be performed by the application. Bulk Synchronous Parallel [205] (BSP) is a less constricting, yet still structured style of programming where programs are written in alternating phases of computation and communication, with the communication phase acting as a barrier synchronization between adjacent computation phases. Because BSP programs do not perform any communication during each computation phase, if each process took a checkpoint during the same computation phase, there would be a guarantee that no messages would cross the resulting restart line. Some checkpointing systems that specifically target BSP applications include the checkpointers for the InteGrade [70] Grid Middleware system and the BSPLib [106] Parallel Programming Library.

The popularity of embarrassingly parallel, master-slave and BSP programs show that the trivial checkpointing protocol (i.e. one that ignores communication)

can be useful for many real applications. However, in practice many problems require complex interactions between computing tasks that are not easily handled by the embarrassingly parallel or master-slave programming models. Furthermore, as applications are being run on more and more processors, the use of global synchronization becomes very expensive, limiting the applicability of BSP-style programs as well. As a result, the approach of relying on the application to identify Uncrossed restart lines ultimately has limited applicability.

### 2.3.1.2 Automatic Protocols

While the above techniques are effective in generating restart lines that are not crossed by any messages, they are undesirable for two reasons.

1. they place a burden on the programmer to identify such restart lines, and
2. they are limited in the types of applications that they apply to.

In light of this there has been work on both automating the programmer's task and making such restart lines plentiful in any application.

Since the programmer identifies Uncrossed restart lines in code by examining its semantics, this job can be automated via a compiler analysis of the source code. One such analysis is presented in [30], where the parallel application's control flow graph is analyzed and sends are matched to receives (the matching is conservative, with one receive matched to one or more possible sends). While the goal of the paper is to identify checkpoint locations that will generate Late-crossed restart lines, it can easily be extended to produce Uncrossed restart lines. However, since the latter is defined more restrictively, the analysis will need to be more conservative and thus, less useful in practice.

Although compiler analyses like [30] can make it easy for Uncrossed restart lines to be generated from applications, they cannot create such lines where they do not exist. Since this is true for a large number of applications, an alternative way to find Uncrossed restart lines is by appropriately constraining the implementation of the message passing library. Buffered CoScheduling [88] is a technique for implementing message passing libraries where all processes are synchronized multiple times per microsecond. The time periods between synchronizations are called "time-slices" and all communications are broken up and scheduled to happen during such time slices. Because no communication crosses the synchronization points, they can form Uncrossed restart lines and checkpoints may be taken there without any concern about communication. [84] presents BCS-MPI, an MPI implementation that employs the principles of Buffered CoScheduling and [163] presents the design of a checkpoint-based fault tolerant version of BCS-MPI that employs the above idea.

### 2.3.2 Uncoordinated Checkpointing

Parallel systems have two important characteristics that constrain the viable types of checkpointing protocols. One is the fact that communication is generally much more expensive than computation, especially on very large systems. Another is that I/O bandwidth from processors to the reliable storage system is generally limited such that saving the GBs-TBs that make of the state of a large parallel application can take a large amount of time. Uncoordinated checkpointing deals with these constraints by having each task checkpoint its own state without coordinating with any other task. This reduces communication costs and allows processes to record their checkpoints at different points in time. The later has the beneficial

effect of only having a few processors sending checkpoints to reliable storage at any given time, reducing the pressure on the I/O subsystem. Furthermore, the fact that tasks do not need to coordinate during checkpointing frees them to record their local checkpoints wherever is most convenient (e.g. spots in the code with the least amount of state) However, despite the intuitive benefits of uncoordinated checkpointing, the absence of coordination causes many checkpoints to be useless in applications with frequent communication.

The basic difficulty is that the lack of any connection between the application's communication and the timing of each task's checkpoints eliminates any guarantees on the types of restart lines that may be taken in such a protocol. Figure 2.12 provides an example of this. It has three tasks, each of which has taken multiple checkpoints. Because of the placement of the checkpoints relative to each task's communication calls, the restart line containing each task's latest checkpoint ( $C_{0,2}$ ,  $C_{1,3}$  and  $C_{2,3}$ ) is Criss-crossed. As discussed in Section 2.2, this requires the checkpointing system to record message data and the outcomes of non-deterministic events, which may be undesirable.

If we do not wish to use Criss-crossed restart lines (and the vast majority of checkpointing literature does not), we need to roll some tasks further back to earlier checkpoints in the hope that they will result in a more desirable restart line. In Figure 2.12 the latest non-Criss-crossed restart line is the Late-crossed line ( $C_{0,1}$ ,  $C_{1,1}$ ,  $C_{2,2}$ ). Restarting from this type of line requires the checkpointing system to record message data, which again may be considered too high a cost. If this is the case further rollbacks may be required, which in our example would cause tasks to roll back to the Uncrossed restart line ( $C_{0,1}$ ,  $C_{1,1}$ ,  $C_{2,1}$ ).

Called the "domino-effect" [179], this phenomenon can be problematic in ap-

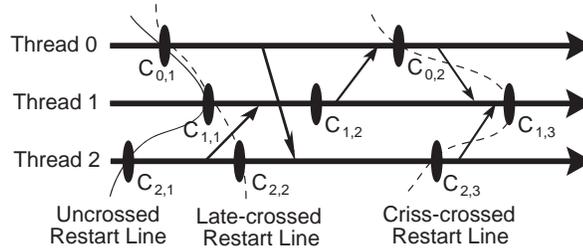


Figure 2.12: Example of the domino effect

plications with frequent communication because of the large numbers of rollbacks that may need to be performed by each task. While any choice of local checkpoint from each task will form a valid Cross-crossed restart line, it is possible that the most recent Late-crossed, Early-crossed or Uncrossed restart line is at the very beginning of the application’s execution. The result is longer roll-backs and potentially many useless local checkpoints that will never be part of any valid restart line. Furthermore, since it is not possible to determine whether a given local checkpoint will be useful when it is recorded, uncoordinated checkpointing protocols must maintain multiple checkpoints per task until they are sure that a given local checkpoint is no longer needed.

Despite its drawbacks, uncoordinated checkpointing remains a useful tool that has been the focus of a significant amount of research. The bulk of it focuses on (1) identifying the latest valid restart line that all processes may roll back to and (2) garbage collecting old local checkpoints that are no longer needed for restart. While most of this work deals with Late-crossed restart lines, similar techniques can be applied to the other types.

In approaches like [43] and [211] valid restart lines are identified off-line, after a failure has happened. While these papers differ in algorithmic details, both work in essentially the same way. The execution of each application task is broken up by its local checkpoints into intervals, as discussed in Section 2.1. Whenever

a message is sent from one task to another, the interval of the receiving task is recorded as depending on the interval of the sending task. When the application is rolled back, all tasks send their dependence information to a single task, which then uses a variant of the following rollback algorithm to compute the most recent Late-crossed restart line.

Let  $n$  be the number of tasks in the application and  $R$  be the restart line that all tasks will roll back to. It is initialized to  $(C_{0,c_0}, \dots, C_{i,c_i}, \dots, C_{n,c_n})$ , where  $c_i$  is the checkpoint number of the most recent checkpoint on task  $i$ . If this restart line is crossed by an early message from task  $i$  to task  $j$  then task  $j$ 's current checkpoint in  $R$  is called an "orphan" and  $j$  is rolled back to the next earlier checkpoint. ( $R \leftarrow (C_{0,c_0}, \dots, C_{j,c_j-1}, \dots, C_{n,c_n})$ ). This process is repeated until  $R$  is no longer crossed by any early message. Since this is true for the trivial restart line  $(C_{0,0}, \dots, C_{n,0})$ , the algorithm is guaranteed to terminate. It is also guaranteed to find the most recent Late-crossed restart line. A variant of this algorithm that finds the latest Uncrossed restart line can be trivially derived from the above algorithm. Work such as [209] is a variant of the above protocols that uses similar ideas to garbage collect checkpoints that precede the most recent valid restart line and will therefore never be needed in the future.

Because most work on uncoordinated checkpointing tries to create Late-crossed restart lines, it becomes necessary to record the data of late messages so that these messages can be replayed on restart. Since tasks record local checkpoints independently of each other, any message may potentially be late relative to some restart line. As such, the data of all messages (or enough information to recompute it) needs to be saved in stable storage in order to make it possible to restart from such restart lines. Since this message data is only needed for restart, it only

needs to be maintained for messages that may cross some restart line that may be rolled back to. As such, an extra protocol is needed to garbage collect the data of messages that are no longer useful. A simple protocol would examine the most recent valid restart line and prune the data of all messages received before it since such messages may not be late relative to this restart line or any restart line that happens afterwards.

### 2.3.3 Quasi-Synchronous Checkpointing

While uncoordinated checkpointing makes it easy to decide when to checkpoint each task, the domino effect introduces notable complexity in finding a valid restart line and more importantly, may cause tasks to roll very far into the past. This costs us time and resources in two ways. First, the longer rollbacks cause restarts to take longer, which can be very costly in low-reliability environments. Second, many local checkpoints taken by tasks may end up not fitting into any valid restart line, meaning that taking these checkpoints is a waste. Since the lack of inter-task coordination makes it impossible to tell at the time of a checkpoint whether the checkpoint will be useful or not, one alternative is to force tasks to take additional checkpoints to ensure that no checkpoint taken by any task is wasted, thus creating more valid restart lines to roll back to.

Quasi-synchronous checkpointing (also known as Communication-induced Checkpointing) uses this insight to extend uncoordinated checkpointing to ensure that no wasted checkpoints are taken. In order to avoid having to do any explicit checkpoint coordination (i.e. by sending additional control messages), research in quasi-synchronous checkpointing has developed an elegant theory that allows it to determine whether a checkpoint must be forced on a given task based on the pattern of

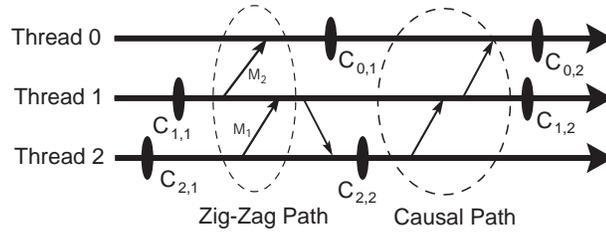


Figure 2.13: Examples of zig zag and causal paths

application sends and receives executed by the task as well as any information that may be piggybacked on these messages. Like uncoordinated checkpointing, quasi-coordinated checkpointing protocols generally aim to create Late-crossed restart lines, although they can be extended to Uncrossed restart lines as well.

The core of the theory of quasi-synchronous checkpointing is in the idea of "zig-zag paths" or "z-paths". A z-path from checkpoint interval  $I_{t,d}$  to interval  $I_{s,e}$  is a sequence of messages  $\langle M_1, \dots, M_k \rangle$  such that

- Message  $M_1$  is sent from task  $t$  in interval  $I_{t,d}$ .
- Message  $M_k$  is received by task  $s$  in interval  $I_{s,e}$ .
- For every pair of messages  $M_i$  and  $M_{i+1}$ , if  $M_i$  was received by task  $t$  during internal  $I_{t,d}$  then  $M_{i+1}$  was sent by task  $t$  during  $I_{t,d}$  or a later interval.

Figure 2.13 shows an example of a z-path (let it be  $\langle M_1, M_2 \rangle$ ). Although  $M_1$  is received by task 1 after it sends  $M_2$ , these messages make up a z-path because  $M_2$  was sent in the same interval as  $M_1$ . This is in contrast to the more intuitive "causal paths" or c-paths (also shown in Figure 2.13), where for each pair of adjacent messages  $M_i$  and  $M_{i+1}$ , if  $M_i$  is received by task  $t$ ,  $M_{i+1}$  is sent by task  $t$  at a later point in time. Note that every c-path is a z-path but not the other way around.

If there exists a z-path from checkpoint  $C$  to checkpoint  $C'$ , we denote this via

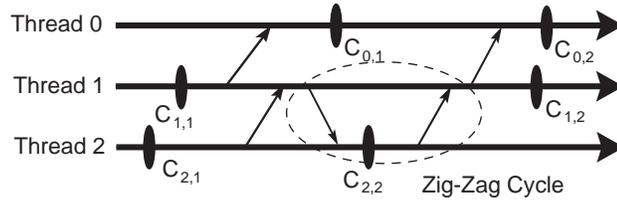


Figure 2.14: Examples of a zig zag cycle

$C \vec{z} C'$ . Similarly the existence of a c-path from  $C$  to  $C'$  is denoted by  $C \vec{c} C'$ . Note that while  $C \vec{c} C$  is not possible as it would correspond to a checkpoint that causally precedes itself,  $C \vec{z} C$  is in fact possible. Figure 2.14 shows an example of such a "zig-zag cycle" or "z-cycle", where  $C_{2,2} \vec{z} C_{2,2}$ .

We can see why z-paths are interesting by looking at Figure 2.13. First examine the c-path  $C_{2,2} \vec{c} C_{0,2}$ . As such, any restart line containing both of these checkpoints must be crossed by an early message, which is exactly what uncoordinated checkpointing was trying to avoid. To understand why this is true in general, let  $C \vec{c} C'$ , via the c-path  $\langle M_1, \dots, M_k \rangle$  that spans tasks  $t_1, \dots, t_{k+1}$ . Suppose a restart line contains both  $C$  and  $C'$ . Since  $M_k$  was received by task  $t_{k+1}$  before checkpoint  $C'$ , initially we know that the c-path crosses  $t_{k+1}$  before the restart line (i.e  $M_k$  is received by  $t_{k+1}$  before its restart line checkpoint). Now, which checkpoint can we choose from task  $t_k$  to appear in the restart line? It must be either before message  $M_k$  was sent by  $t_k$  or afterwards. If before, then  $M_k$  is early relative to this restart line since it is sent after the restart line and received before. If after, then we maintain our initial condition: the c-path crosses  $t_k$  before the restart line. This reasoning is repeated along the entire *c-path*. If we get to  $t_1$  without showing that some message is early then we know that our condition still holds: the c-path crosses  $t_2$  before the restart line. Since we know that  $M_1$  was sent by  $t_1$  after it took checkpoint  $C$ ,  $M_1$  must be early relative to the restart line.

z-paths are a generalization of c-paths for which the same argument holds. This is because the argument above applies to how messages are ordered relative to checkpoints rather than each other. Since  $C \xrightarrow{z} C'$  implies that any restart line that contains checkpoints  $C$  and  $C'$  must be crossed by an early message, the analysis of z-paths becomes important in trying to develop protocols that can avoid such early messages. In particular, z-cycles of special interest because it can be shown that if a checkpoint is involved in a z-cycle, any restart line containing it will be crossed by an early message. As such, these checkpoints are useless and should not have been taken in the first place.

Because z-paths and z-cycles can make it difficult to find a Late-crossed restart line or cause tasks to take useless checkpoints, it becomes imperative to break these paths and cycles and avoid the problem. Because z-paths and z-cycles are defined in terms of the checkpoint intervals they are taken in, they can be broken by forcing tasks to take additional checkpoints, which break up the intervals more finely. Because vector clocks [140] [85] [129] can be used to easily track c-paths online [51] [108] [109] [210], it is easily possible to avoid early messages induced by such paths by having each process take a local checkpoint immediately before it receives a message that would create such a c-path. Dealing with z-cycles and z-paths that are not also c-paths (called "non-causal" z-paths) is much more difficult and has spawned a great variety of protocols.

Following [138], quasi-synchronous checkpointing protocols can be divided into three types: Strictly Z-Path Free (SZPF), Z-Path Free (ZPF) and Z-Cycle Free (ZCF).

### 2.3.3.1 Strictly Z-Path Free

These protocols make sure that enough forced checkpoints are taken to ensure that there are no z-paths in any execution. The MRS protocol [184] [26] makes sure that no receive can follow a send within a given checkpoint interval by forcing a checkpoint before this can happen. As a result, in each interval all receives precede all sends, making non-causal z-paths impossible. Simpler variants of above protocol include checkpointing after every send, checkpointing after every receive and checkpointing after every send or receive [210] and work in the same fashion.

### 2.3.3.2 Z-Path Free

Although SZPF protocols avoid all possible z-path, this is an unnecessarily strong condition. Since tracking and dealing with c-paths is relatively easy, the only thing that we need forced checkpoints for is the elimination of z-paths for which there exist no corresponding c-paths (i.e. we want to ensure that if  $C \vec{z} C'$  then  $C \vec{c} C'$ ). The class of protocols that enforce this condition is known as Z-Path Free. While having most of the advantages of SZPF protocols ZPF protocols force fewer checkpoints since they need to satisfy a weaker condition.

The Fixed-Dependency-Interval protocol [210] uses vector clocks to track the checkpoints that have been taken by each task. Each task maintains the checkpoint number of each task's latest known checkpoint, forming a checkpoint number vector. Whenever a task checkpoints, it updates its own checkpoint number in its vector. All outgoing messages have the task's current checkpoint vector piggy-backed on top of them and whenever a message arrives, its checkpoint vector is read and the receiver task's vector is updated. The Fixed-Dependency-Interval protocol uses vector clocks in the following way: during each checkpoint interval,

if a task receives a message with a checkpoint vector that has newer checkpoints than the receiver task's checkpoint vector, the receiver task takes a checkpoint. The Fixed-Dependency-After-Send protocol [210] is generalization of the above protocol in that it may only force a checkpoint after a send has occurred during a given checkpoint interval.

### 2.3.3.3 Z-Cycle Free

While the existence of z-paths can make it impossible for one checkpoint to be in a Late-crossed restart line with some other checkpoint, it is not hard to find a usable Late-crossed restart line that uses one of the checkpoints but not both. As such, an even more relaxed checkpoint quality condition would be: make sure that all checkpoints are useful. In other words, make sure that if a checkpoint is taken, there exist some valid Late-crossed restart line that the checkpoint can be a part of. It can be shown [148] that all checkpoints are useful if and only if there exist no z-cycles in the application's messages, which is exactly what the Z-Cycle Free category of protocols tries to ensure.

The protocol in [51] tries to form valid restart lines of each task's  $i^{th}$  checkpoint (i.e. each task's  $i^{th}$  checkpoint belongs to the  $i^{th}$ 's Late-crossed restart line). It does this by piggybacking the current checkpoint interval number on every outgoing message. When a message is received with a higher checkpoint number, a local checkpoint is forced. It can be easily shown that this ensures that each checkpoint is useful. Suppose that some restart line  $i$  is crossed by an early message from task  $s$  to task  $r$ . As such, the checkpoint number piggybacked onto this message must have been  $i$ . But since it was received by  $r$  before it took its checkpoint, the message must have been received by  $r$  after  $r$  took its checkpoint  $i$ . As such,

$r$ 's checkpoint that is part of this restart line must be have a checkpoint number  $> i$ , meaning that it cannot be part of global restart line  $i$ . [137] is a variant of the above that can create restart lines from local checkpoints that do not have the same index number.

Note that neither approach explicitly tracks  $z$ -cycles. Instead, both enforce stricter properties that themselves imply  $z$ -cycle freedom. As such, they may force more checkpoints than are strictly necessary under the  $z$ -cycle freedom definition.

#### 2.3.3.4 Performance Considerations

While quasi-synchronous checkpointing offers the promise of Late-crossed restart lines with no additional synchronization and a relatively small communication cost (associated with piggybacking data on messages), it has the drawback of forcing potentially large numbers of checkpoints in order to maintain its guarantees. [180] experimentally examines the performance of three such protocols and find them to record between 2x and 10x forced checkpoints relative to the number of regular checkpoints, with the number of forced checkpoints rising approximately linearly with the number of processes. The result is that checkpointing becomes very expensive and unscalable. Furthermore, checkpoints were taken with unpredictable frequency, making it difficult to appropriately allocate resources for checkpointing. The paper proposes a variation on quasi-synchronous checkpointing that aims to reduce the number of forced checkpoints by not taking a regular checkpoint if there has been a recent forced checkpoint. While the number of forced checkpoints is generally reduced by a 2x, it is still high.

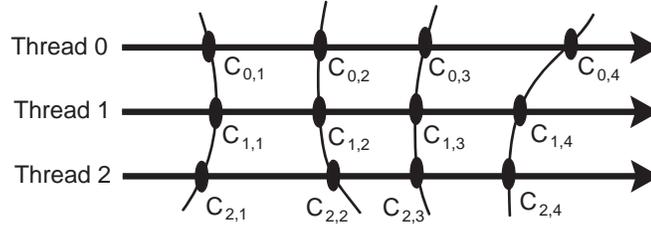


Figure 2.15: Blocking checkpointing of an application

### 2.3.4 Coordinated Checkpointing

In light of the difficulties encountered by coordination-free checkpointing, the alternative approach to checkpointing is to coordinate the checkpointing activities of different tasks so as to create restart lines of a certain type by construction. In particular, these protocols choose a given type of restart line (usually Late-crossed) and schedule the times when each task records its  $i^{\text{th}}$  checkpoint to ensure that the resulting restart line (containing the  $i^{\text{th}}$  checkpoint taken by each task) is of the chosen type. This may require forcing some tasks to take their  $i^{\text{th}}$  checkpoint earlier than they may have wished to. The result (shown in Figure 2.15) is an execution that is divided by discrete, non-intersecting restart lines, with the  $i^{\text{th}}$  restart line being defined as  $(C_{0,i}, \dots, C_{n,i})$ . Messages that are past/future relative to all restart lines are called *Intra-interval* messages since their `send` and `receive` operations happen within the same checkpoint interval.

The advantage of coordination is that the more orderly creation of checkpoints guarantees the existence of recent usable restart lines. On the other hand, the additional communication has its own cost and tasks may be forced to checkpoint themselves at less than optimal locations (although this forced checkpoint effect is not as severe as with quasi-synchronous checkpointing). Furthermore, most coordinated checkpointing protocols checkpoint all the tasks in a relatively short period of time, which can put pressure on the I/O system.

In prior literature there exist two types of coordinated checkpointing protocols, blocking and non-blocking, described in the following sections.

#### 2.3.4.1 Blocking Coordinated Checkpointing

As discussed in Section 2.3.2, the simplest way to make sure that a given restart line is not crossed by messages is to checkpoint at a barrier. Blocking coordinated checkpointing extends this insight to applications that do not have points in their executions that can be identified as being free of communication. The idea is that if we want to checkpoint the parallel application, we will tell each task to record its own checkpoint and then stall until two conditions are satisfied:

1. all tasks have recorded their local state, and
2. all messages that were sent by tasks before they checkpointed have arrived at their destination tasks

The result is a set of checkpoints that is guaranteed to from a Late-crossed restart line. However, since only these checkpoints were coordinated in this fashion, the same is not guaranteed to hold for other sets of checkpoints.

**2.3.4.1.1 Sync-and-stop** The simplest version of the above protocol idea is known as "synch-and-stop" [167]. The basic sync-and-stop protocol, similar to those used in [201] [170] [171] [60] [132] [187] [218], is shown in Figure 2.16. The messages with solid lines are application messages and the ones with dashed lines are the protocol's messages. Checkpointing begins when some coordinator task (task 0 in this example) chooses to start a checkpoint. The coordinator first stops its application, sends a `startChckpt` message to every other task and proceeds

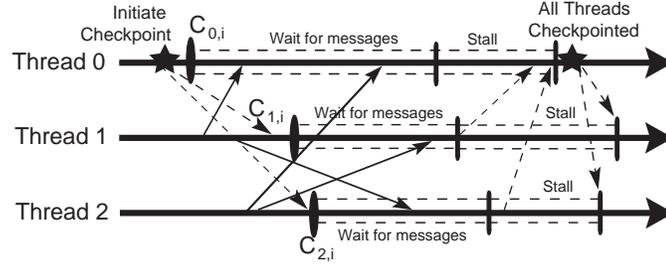


Figure 2.16: Examples of a blocking coordinated checkpoint

to record its checkpoint. When a task receives the `startChckpt` message, it also stops the execution of its application and immediately takes a checkpoint.

After a task (coordinator or any other task) has finished recording its checkpoint it does not resume executing the application. Instead, it waits for all the messages that it sent before it took its checkpoint to arrive as their destinations (via one of a variety of possible mechanisms). Messages that arrive at a task during this time period are logged by that task since they must be late relative to this restart line (i.e. sent before the sender’s checkpoint and will be delivered to the receiver task’s application after the receiver’s checkpoint). When a task is sure that all messages that were in-flight to it during checkpoint have been delivered, it sends a `latesDone` message to the coordinator. When the coordinator has received `latesDone` from every task it knows that all tasks have recorded their checkpoints and all late messages have been delivered. The checkpoint image on disk is now complete and can be committed. As such, the coordinator sends a `resumeApp` message to every task and when a task receives such a message, it resumes executing its application.

Sync-and-stop is a very simple, easy to implement protocol. Furthermore, in some scenarios it has also been shown to be relatively efficient [170] [171] [60] (i.e. it cost significantly less than saving the checkpoint to reliable storage). However, because sync-and-stop forces all tasks to block while a checkpoint is being taken, there are a number of situations where it becomes an obviously bad choice. One

prime example is distributed systems, where the cost of communication is high, causing tasks to block unnecessarily long while message traverse high-latency wide area networks. Another example is extremely scalable systems that may contain thousands of processors. Because of its need for tight synchronization, sync-and-stop would result in very large amounts of communication on such systems. Finally, the fact that all processes must checkpoint essentially simultaneously puts the maximum amount of possible pressure on the I/O system.

**2.3.4.1.2 Time-coordinated** Since one of the important costs of sync-and-stop is the significant amount of communication it performs in order to schedule its checkpoints, there is clear motivation for a protocol that does blocking coordinated checkpointing but without as much communication. Time-coordinated checkpointing [151] [152] [153] is such an alternative that uses the relative synchronization of the clocks of the different tasks to ensure proper synchronization.

The idea is that if the relative clock drift of the tasks (the amount by which the system clock deviates from real time over a given time period) is bounded, then it is possible for one task to place an upper and lower bound on the current time on every other task. Suppose that checkpoints are set to happen once every  $P$  seconds, when a periodic timer goes off. If a given task's checkpoint timer goes off (i.e.  $P$  seconds have elapsed), it has upper bound on when the timers of other tasks will also go off. The blocking time-coordinated protocol in [152] shows how these bounds can be used to ensure that no late or early messages cross the restart line, as shown in Figure 2.17.

Suppose that task  $t$ 's timer has gone off and it is ready to checkpoint. Given the bounds on the clock drift,  $t$  knows that the timer of any other task  $r$  will go off

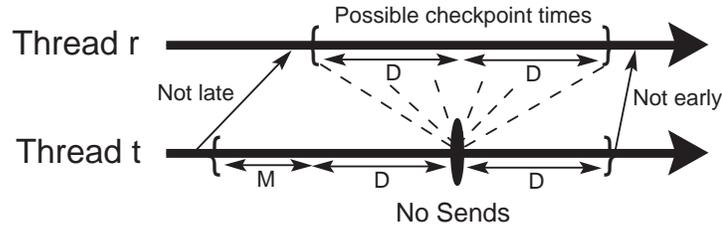


Figure 2.17: Outline of the blocking time-coordinated protocol

no earlier than  $D$  seconds into the past or  $D$  seconds into the future. Thus, in order to make sure that the resulting restart line is not crossed by any early messages  $t$  has to refrain from sending any messages to any other tasks for  $D$  seconds after it begins a checkpoint. This works because any message sent by  $t$  after it took its checkpoint will definitely be sent after all the other tasks took their checkpoints, meaning that it cannot possibly be early.

Making sure that the restart line cannot be crossed by late messages requires us to assume some known upper bound  $M$  on the amount of time it takes for a sent message to arrive at its destination (possible in some networks). If each task then makes sure not to send any messages for  $M + D$  seconds before its next checkpoint timer goes off, it can be sure that no messages that it sends will be late relative to the next restart line. The reason is that any messages sent by task  $t$  before  $t$ 's pre-checkpoint blackout period are guaranteed to arrive at their destinations within  $M$  seconds. Furthermore, the bound on the clock drift guarantees that the destination task's checkpoint timer will go off no earlier than  $D$  seconds before  $t$ 's timer goes off. This makes it possible to generate an Uncrossed restart line at the cost of shutting down all system communication for a short period of time.

Because the assumption of a known bound on message delivery times may be too strong in many situations, the protocol in [152] was extended in [153] to avoid this assumption. The idea behind this semi-blocking protocol is to remove the

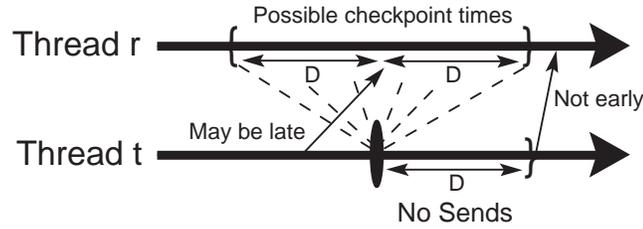


Figure 2.18: Outline of the semi-blocking time-coordinated protocol pre-checkpoint send blackout (Figure 2.18). Since this blackout period allowed us to keep the restart line from being crossed by late messages, this protocol produces Late-crossed restart lines, as opposed to the Uncrossed lines produced by the protocol above. As such, the protocol is further extended to log any late messages. Since [153] focuses on unreliable channels, it assumes that it is part of a layer that sends messages and waits for them to be acknowledged, performs duplicate detection, etc. Thus, to record the data of all late messages, at checkpoint time each task records the data of all messages that it sent before the checkpoint but that were not acknowledged by the time the checkpoint was taken. Since this is too loose a definition, the protocol can end up saving some past messages in addition to real late messages. However, if those messages are resent on restart the duplicate detection functionality in their communication system will automatically drop them.

Since both of these time-coordinated protocols depend closely on the upper bound on the difference between the clocks on different tasks, their performance will suffer over time as different tasks' clocks drift further and further out of synchrony. The reason is that the value  $D$  will grow larger and larger, until the communication blackout periods constitute a large fraction of the application's execution. The solution is a clock synchronization algorithm, variants of which are presented in [152] and [153]. This algorithm is run whenever clock uncertainty grows past a

certain bound and brings it back down to a low value. With real computer clocks having drift on the order of  $10^{-5}$  -  $10^{-6}$  seconds of computer time per seconds of real time, the synchronization protocol needs to be invoked fairly rarely.

[152] experimentally compares the blocking time-coordinated protocol to a non-blocking coordinated protocol similar to the last variant of Chandy-Lamport described in Section 2.3.4.2 and finds that it has lower overhead. [123] compares the blocking and semi-blocking protocols to each other using an analytical model and finds that the semi-blocking protocol is more efficient in most cases.

### 2.3.4.2 Non-Blocking Coordinated Checkpointing

**2.3.4.2.1 Late-crossed Restart Lines** Nonblocking coordinated checkpointing extends its blocking counterpart by removing the requirement for all tasks to block until the global checkpoint has been recorded. As such, it becomes possible for some tasks to have finished their checkpoints while others have not yet begun theirs. The invariant that is maintained in all these protocols is that the restart lines that they generate can only be crossed by late messages. Since this can be achieved without forcing tasks to all checkpoint themselves at the same time, these protocols achieve greater efficiency and scalability than strictly blocking protocols because they waste less application time and put less pressure on the I/O system.

Among the plethora of work on non-blocking checkpointing protocols, the Chandy-Lamport distributed snapshot protocol [59] is the most well-known. The reason lies both in the simplicity of the protocol itself and how well it delineates the job of a non-blocking protocol. The idea is simple: if some task  $t$  takes a checkpoint, then the only way to keep the resulting restart line from being crossed by early messages is to make sure that every other task checkpoints itself before

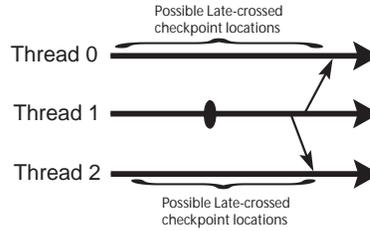


Figure 2.19: Limits on when checkpoints must be taken

it receives a message from  $t$  that  $t$  sent after it took its checkpoint. In Figure 2.19 task 1 has taken a checkpoint and will send messages to tasks 0 and 2. Thus, both of those tasks must take their own checkpoints in the indicated time regions. If they do not, the messages become early relative to the resulting restart line.

The Chandy-Lamport protocol is built around the above intuition. It assumes that the communication channels between tasks are uni-directional and defined a priori (more than one channel may go between each pair of tasks) and that they deliver messages in FIFO order. Furthermore, it assumes that the communication graph of these channels is strongly connected (i.e. every task may communicate with every other task by passing messages through some sequence of other tasks). The protocol works by ensuring that each task takes its  $i^{th}$  checkpoint early enough that no incoming messages may be early relative to the  $i^{th}$  global restart line.

The protocol starts when one or more tasks independently choose to record their  $i^{th}$  checkpoints. When a task takes checkpoint (either by choice or because it was forced to by the protocol), it sends a notification message to every other task to which it has an output channel. Whenever some task  $r$  receives a notification message from another task  $t$ , it knows that  $t$  has checkpointed.  $r$ 's job is to take its  $i^{th}$  checkpoint early enough that no message from any of its neighboring tasks may be early relative to the  $i^{th}$  restart line. The fact that the communications channels are FIFO tell us that any message received by  $r$  from  $t$  after  $r$  received

$t$ 's notification must have been sent by  $t$  after it took its checkpoint. As such, the latest point in time for  $r$  to take a checkpoint and avoid generating early messages is immediately before it sends a message to task  $s$  after  $r$  has received a notification message from  $s$ . It must be forced to do so if it does not choose to do it on its own.

While this takes care of early messages, recall that to restart from a Late-crossed restart line we need to be able to recreate the data of all of its late messages. As such, we need to identify which messages are late. Recall that a late message from task  $t$  to task  $r$  was sent before  $t$  took its checkpoint and received after  $r$  took its checkpoint. Since we assumed that the communication channels are FIFO, this means that all messages received by  $r$  before it received  $t$ 's notification message must have been sent by  $t$  before it took its checkpoint. As such, all messages received by  $r$  from  $t$  after  $r$  took its checkpoint and before  $r$  received  $t$ 's notification message must be late and their data needs to be recorded.

The above considerations give us the full Chandy-Lamport protocol:

- When a task takes a checkpoint (either independent or forced), it sends a notification message to each of its neighbors (task that it has an outgoing channel to)
- When a task receives a notification message, it immediately takes a checkpoint if has not done so already for the current restart line (this is more conservative than the checkpoint forcing condition above)
- If a task receives a message from task  $t$  after it has taken its  $i^{th}$  checkpoint but before it has received  $t$  notification message from  $t$ 's  $i^{th}$  checkpoint, the data of this message is recorded so that it can be played back on restart (since it is a late message)

- No task can take its  $i + 1^{st}$  checkpoint until the  $i^{th}$  restart line has been committed. (the commitment protocol can be any standard agreement protocol)

A method for generalizing this protocol to non-FIFO channels is shown [77]. The Chandy-Lamport protocol uses FIFO assumption to identify what messages were late and early by comparing their arrival times relative to the arrival time of the notification message. Without this assumption, arrival times are useless for this purpose and another mechanism must be found. An alternative solution is to note that in coordinated checkpointing all checkpoints in a given restart line have the same index number (i.e. the  $i^{th}$  restart line is composed solely of each tasks'  $i^{th}$  checkpoints). As such, a message from  $t$  to  $r$  is late relative to the  $i^{th}$  restart line if it was sent from interval  $I_{t,i-1}$  and received in interval  $I_{r,i}$ . Similarly, the message is early if it is sent in interval  $I_{t,i}$  and received in  $I_{r,i-1}$ . Thus, an alternative mechanism would attach to each message the interval number from which it was sent. When a message is received by task  $r$ , the receiving task knows whether it is late, early or neither by comparing its own interval number  $I_{r,j}$  to the interval number attached to the message  $I_{t,k}$  as follows:

- The message is Late if  $k < j$ .
- The message is Early if  $k > j$ .
- The message is Intra-interval if  $k = j$ .

This protocol can be optimized further by noting that because coordinated checkpointing does not allow restart lines to cross, it is only possible for a task to receive messages from the last, current and next intervals. Thus, instead of attaching entire interval numbers to messages, it is sufficient to attach just the last two bits. This can be reduced to just the last bit if a new global checkpoint is not

allowed to begin until all threads know that the most recent checkpoint has been completed.

[40] is an example of a system that uses a variant of the Chandy-Lamport protocol. [77] experimentally compares their variant of Chandy-Lamport to the Uncoordinated checkpointing protocol of [43]. They find that the two approaches have comparable performance overhead on a 16-processor cluster with 2-minute checkpointing intervals. Only two codes showed notable differences in performance. On one code Chandy-Lamport performed worse because the coordinated checkpoints were all sent to disk at roughly the same time, putting pressure on the I/O subsystem. Another code showed higher overhead under Uncoordinated checkpointing. This code has a highly coordinated communication pattern, meaning that if some processes independently slow down while taking checkpoints, this slows the entire computation down. A similar effect was noted in [124] when analyzing the effects of daemon activity on the performance of programs running on the ASCI Q [2] supercomputer.

**2.3.4.2.2 Criss-crossed Restart Lines** While most work in non-blocking coordinated checkpointing has focused on creating late-crossed restart lines, little attention has been paid to coordinated protocols for creating criss-crossed lines. As discussed in Section 2.2.1, such restart lines occur naturally in application-level checkpointing and there is clear need for coordinated protocols that can handle them. While such protocols do not exist in prior literature, Section 2.5 presents protocols that does exactly this.

## 2.4 Message Logging Protocols

Message Logging protocols are defined as those that create restart lines that may contain the current state of some process(es). This is in contrast to restart lines generated by checkpointing protocols, which must contain only previously checkpointed states. The principal advantage of message logging protocols over checkpointing protocols is that the rollback of a single task does not necessarily require the rollback of any other task (additional rollbacks may happen in some protocols). This is important for two reasons:

- The execution of other tasks is not significantly slowed down rollbacks (it may be if the application is tightly synchronized)
- The requirement made by checkpointing protocols that all tasks must roll back to a prior checkpoint if any one needs to be rolled back (e.g. because it failed) does not scale well as the probability of failure rises and as the number of tasks becomes large. Message logging protocols scale much better since they do not have such a requirement.

As discussed above, realistic message logging protocols must generate criss-crossed restart lines. This means that in addition to periodically checkpointing the state of each task, the data of all messages must be saved and the outcomes of all non-deterministic events must be recorded. This constant overhead, even in the absence of failures, is the primary drawback of message logging protocols. It can be particularly severe in high-communication applications because processors quickly run out of memory in which to store message logs.

Existing message techniques fall into three categories: pessimistic, optimistic and causal, with the primary difference between the classes of protocol lying in how

they log the outcomes of non-deterministic events. Pessimistic logging ensures that all such events are reliably stored at all times and no task state may depend on non-deterministic event that has not been reliably logged and may happen differently if some other task restarts. As such, pessimistic protocols ensure that the rollback of one task can never cause another task to also roll back. Since this guarantee can have a high performance cost, optimistic protocols are not as precise about their logging of non-deterministic events, making it possible for some tasks that were not rolled back to be forced to roll back because their state depends on the outcome of some non-deterministic event that was lost due to insufficient logging. Causal logging presents a balance between pessimistic and optimistic logging by spreading the storage of non-deterministic events across the tasks that may depend on them. As such, it provides similar reliability guarantees to pessimistic logging while providing the improved efficiency of optimistic logging.

### **2.4.1 Pessimistic**

The guarantee provided by pessimistic message logging is that at all times there is never a task the state of which depends on the outcome of a non-deterministic event that could get lost as a result of a task rollback. The basic pessimistic message logging protocol rigidly implements this definition. Whenever a task sends a message, it saves it in its volatile memory. Whenever it performs a non-deterministic event, it saves its outcome in stable storage and does not send any more messages until it is sure that the event is reliably stored. When some task rolls back, all other processes resend to it all the messages that it received since its most recent checkpoint (these became late because of the rollback). It then reads all of its old non-deterministic decisions from the log on reliable storage. As it resumes its

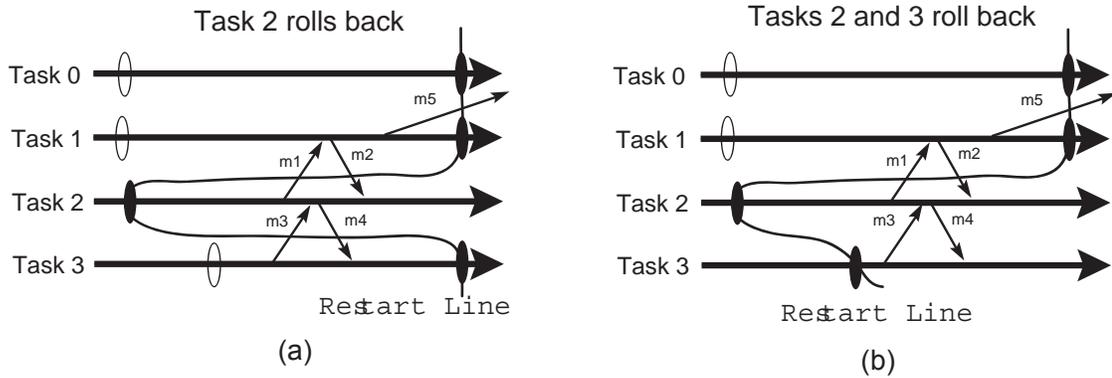


Figure 2.20: Sample message logging restart lines

execution, it is guaranteed to deterministically recreate its pre-rollback state by

- repeating its original non-deterministic decisions,
- re-receiving all of its old messages and
- allowing all messages that it had sent before its rollback (these became early as a result of the rollback) to be resent but having ignoring the non-rolled back tasks ignore them..

An example of this is shown in Figure 2.20(a), where task 2 has rolled back. This rollback makes messages  $m_1$  and  $m_4$  early and messages  $m_2$  and  $m_3$  late. When task 2 restarts, task 1 must resend to it message  $m_2$  from its volatile log and task 3 must resend message  $m_3$ . As task 2 restarts and deterministically reexecutes from its prior checkpoint, it will try to resend messages  $m_1$  and  $m_4$ . These must either be suppressed or ignored. When task 2 has finished its restart procedure, it will have restored itself to its state at the time before it rolled back and no other task will depend on any non-deterministic decision the outcome of which was lost due to the rollback.

Figure 2.20(b) shows what happens when tasks 2 and 3 roll back together. In this case messages  $m_3$  and  $m_4$  are no longer late and early but rather future. As such, on restart when tasks 2 and 3 send these messages, they will not be ignored by their recipients because their recipients were also rolled back. Since both tasks are guaranteed to re-execute deterministically, this is equivalent to these tasks resending these messages from their logs, except that in this case, the data of these messages was recomputed.

The above protocol ensures the basic invariants of pessimistic message logging but does not do so efficiently. The main bottleneck is that the outcomes of non-deterministic decisions must be stored in reliable storage before the next message may be sent. Since access to reliable storage is typically slow, this can significantly reduce the performance of pessimistic logging. One possible solution is presented in [119], a pessimistic message logging protocol that can tolerate the failure of a single node and allows non-determinism only in the arrival order of messages. The insight in [119] is that if we limit ourselves to only tolerating the failure of a single task at a time, the task's non-deterministic decisions can be stored in the volatile memory of any other task.

[119]'s protocol is similar to the one described above. When a message is sent by task  $t$  to task  $u$ , its data is saved in task  $t$ 's volatile memory. When  $u$  receives the message, it replies with an acknowledgement message that contains the message's arrival order.  $u$  then continues working but does not send any messages until it receives an acknowledgement from  $t$ . This is critical because it ensures that no other task's state will depend on  $u$ 's non-deterministic decisions until they are in some safe location. When  $t$  receives its non-determinism acknowledgement, it adds this non-determinism data to its log and replies to  $u$  with an acknowledgement.

If a task  $v$  rolls back, all other tasks send to  $v$  messages that  $v$  received from them since its last checkpoint as well as the outcomes of all of  $v$ 's non-deterministic decisions from the same time period. Since each task's non-deterministic decisions are guaranteed to be stored on other tasks, any one task can roll back without the loss of any non-deterministic decisions.

This protocol can be extended to deal with non-deterministic events other than message arrival order as follows: once a task  $u$  has performed a non-deterministic event, it sends a message to some other task  $t$ , informing it of the event's outcome. It may not send any messages until it receives an acknowledgement from  $t$  that it has added the outcome of this event to its log. Another alternative may be to piggyback on top of each outgoing message the outcomes of preceding non-deterministic events. This is roughly what is done in causal piggybacking protocols.

Another possible extension would be to enable the above protocol to tolerate more faults. In this extended protocol new tasks send out additional messages notifying more other tasks about the outcomes of their non-deterministic events and refraining from any sends until they have received acknowledgements from all these tasks.

## 2.4.2 Optimistic

Whereas pessimistic message logging ensures that the state of no process depends on the outcome of a non-deterministic event that can be lost due to a rollback, optimistic protocols relax this guarantee and instead try to detect cases where task state depends on lost events and rolls such tasks back. For an example of how optimistic protocols differ from pessimistic ones, consider Figure 2.21. In this scenario, task 2 receives message  $m_1$ , which is a non-deterministic event. It then

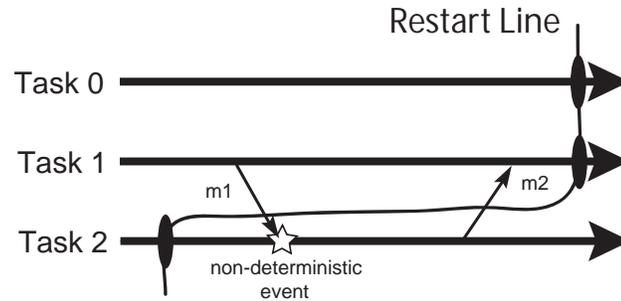


Figure 2.21: Sample optimistic execution

sends a message to task 1, which causes task 1's state to depend on this event. Because an optimistic protocol will not guarantee that  $m1$ 's receive order will be logged to reliable storage, when task 2 rolls back it is possible that this information will be lost with this rollback. This means that it may not be possible to restore the state of task 2 to a state that is consistent with task 1's current state, implying that task 1 must be rolled back to a prior checkpoint.

In short, pessimistic protocols ensure that for all messages, if a message becomes early then the non-deterministic events that it may depend on are guaranteed to be logged somewhere other than the message's sender task. In contrast, optimistic protocols provide no such guarantee. As such, if the outcome of a non-deterministic event that an early message depends on is lost, the message's receiver task must also be rolled back, potentially causing all tasks to roll back to prior checkpoints, reducing the message logging protocol to a checkpointing protocol. In light of this, the main jobs of an optimistic protocol are (i) minimizing the probability that the outcome of a non-deterministic event that an early message depends on is lost due to a rollback and (ii) provide adequate dependence tracking to ensure that this loss is detected and the receiver of this early message is also rolled back. While the first job is rarely studied at anything more than an intuitive level, the second job has been studied extensively in prior research.

In addition to the pessimistic protocol described above, [119] presents an optimistic variant of the above protocol that can also tolerate a single failure and allows no non-determinism besides message arrival order. While in the pessimistic protocol receiver processes acknowledge message receives and do not send messages until their acknowledgements have been acknowledged, the optimistic protocol dispenses with these latter acknowledgements and allows receivers to send messages whenever they choose (the receiver still does send the message's reception order back to the sender). This can cause task states to depend on unlogged non-deterministic events. These dependencies are tracked by piggybacking on top of every outgoing message the total number of messages received by the sender thus far. Each task  $t$  then records a vector  $depR_t[]$  containing the highest such number that it received from each of its neighbors. Thus,  $depR_t[u]$  is the total number of non-deterministic events from task  $u$  that task  $t$  depends on.

On restart, the optimistic protocol tries to restore the state of the rolled back task (task  $t$ ) the same way as the pessimistic protocol. When the task has finished restarting (i.e. recomputing, and deterministically re-receiving messages), it can compare the number of messages that it believes that it has received (i.e. the number of receive events logged by other tasks) to  $depR_u[t]$ , the number actually received by each task  $u$ . If some other tasks depends on more of  $t$ 's message receive events than were actually logged, these tasks are rolled back and restarted, one at a time, using the above restart scheme. Eventually, it is possible for all tasks to restart from their past checkpoints, in which case the details of the resulting restart line depend on the protocol used to schedule task checkpoints.

[199] presents a protocol that is an optimization of the optimistic message logging protocol presented in [136] and [46]. Like the protocols above, these protocols

limit non-determinism to include only the reception order of messages. The basic protocol works as follows. Each message sent by each task is given a unique ID equal to the number of messages sent by this task thus far. Each task  $t$  maintains a dependence vector  $depS_t[]$ , with one entry for each other task  $u$ , containing the ID of the most recent unlogged message from  $u$  that  $t$ 's state depends on. When a task sends a message, it piggybacks its dependence vector on this message. Whenever a message is received, the receiving task (i) updates its own vector to be the element-wise maximum of its pre-receive value and the vector attached to the received message and (ii) asynchronously logs the data of the received message and the receive order of this message and continues computing. When a task rolls back, the log is read to determine the number of messages from the rolled back task that were recorded (i.e. their data and receive order) and declares any later messages lost. Any task that depends on these lost messages according to its  $depS_t[]$  must roll back to a prior checkpoint. Since it is possible for the rollback of one task to cause the rollbacks of other tasks, it is imperative for each task to maintain multiple old checkpoints to ensure that it can always find a prior checkpoint that it can roll back to no matter which task rolls back. The original protocol includes a garbage collection mechanism to erase old checkpoints when they are no longer needed by lazily informing tasks when messages are finally logged by appropriately updating their  $depS_t[]$  vectors.

The optimization presented in [199] removes the need for reliable storage and instead logs message data in the volatile memories of the tasks themselves. This works the same way as in the pessimistic protocol from [119]. If task  $t$  rolls back, its previously received messages are resent to it. If another task  $u$  also rolls back while  $t$  is restarting and loses some of the messages required by  $t$ , it will be restarted and

in the process of deterministically recreating its own state will resend to  $t$  any lost messages. Logging of non-deterministic events is optimized by using a predictor to predict the arrival order of messages. The arrival order is only logged if the predictor makes the wrong prediction.

### 2.4.3 Causal

The idea behind causal message logging is that if task  $t$  depends on a non-deterministic decision made by task  $u$ , then the safest thing for  $t$  to do is store this decision locally in case  $u$  rolls back. This way, if  $u$  rolls back,  $t$ 's state will not need to be rolled back because it can simply send to  $u$  the outcomes of all of  $u$ 's decisions that  $t$  cares about. As other tasks come to depend on  $t$ 's state, it may need to propagate to them all the non-deterministic decisions that its own state depends on, thus causally linking the states of all tasks to the non-deterministic decisions that they depend on.

The Family-Based Logging protocol [34](FBL) is an instance of this idea that can deal with the rollback of a single task. Message senders log the data of sent messages in their volatile memories. Whenever a non-deterministic event occurs on task  $t$ , its outcome is piggybacked on all messages sent by  $t$ . When such a message is received by task  $u$ , its piggybacked non-deterministic outcomes are stored in  $u$ 's volatile memory (critical since  $u$ 's state now depends on these events) and  $u$  responds with a message acknowledging the receipt these outcomes.  $t$  keeps piggybacking the outcome of a given non-deterministic event on outgoing messages until it receives an acknowledgement that the event has been logged by some other task. If any task rolls back, all other tasks resend to it all of the messages that is received before its rollback as well as the outcomes of all the non-deterministic

decisions that other tasks depend on. This allows the task to deterministically recreate its state up to the last non-deterministic decision that any other task's state depends on.

Manetho [78] is a causal logging protocol that can tolerate the rollback of any number of tasks. Like FBL, Manetho requires senders to log the data of outgoing messages in volatile memory. The novel bit is how Manetho deals with non-deterministic events. Time on each task is broken up by the receives of messages and other non-deterministic events. The time period between two such events is called a "state interval". Manetho relates the state intervals on different tasks via an "antecedence graph" (AG), which records the happens-before relationship between state intervals and the outcomes of the non-deterministic decisions that start each state interval. Each task maintains the AG that corresponds to its current state, including all state intervals that it depends on and the happens-before relationships that lead from those state intervals to its current state. These task AG's are updated by having each task piggyback its current AG on top of each outgoing message (in practice, only the difference between the current and the last version of the AG sent is actually piggybacked) and having the receiver incorporate the incoming AG into its own AG. This mechanism is quite similar to the dependence vector mechanism used in [199] above.

When a task  $t$  rolls back, it notifies all other tasks. When task  $u$  receives this notification, it looks at its AG to find the latest state interval on  $t$  that  $u$  depends on and drops any incoming messages that depend on any later state interval on  $t$  (Manetho makes no assumptions/guarantees about reliable message delivery). All tasks then send back to  $t$  their AG, which are merged by  $t$  into a combined AG that contains the outcomes of all of  $t$ 's non-deterministic decisions that any other task

depends on (if no task depends on a decision, then it will not appear in its AG).  $t$  can then deterministically re-create its state upto the last state interval that any other task's state depends on and then continue executing non-deterministically after that.

[35] examines both Manetho and FBL and proposes a continuum of protocols that can tolerate the rollbacks of any  $f$  tasks (Manetho provides  $f=\text{all}$  while FBL provides  $f=1$ ). It proposes a protocol based on the Dependency Matrix data structure that ensure that at all times the outcome of every non-deterministic event is held by  $\min(f + 1, dep)$ , where  $dep$  is the number of tasks that depend on that event at that point in time. This ensures that if upto  $f$  tasks roll back, either (i) there exist tasks that depend on the non-deterministic events that happened on the rolled back tasks and those events are not lost or (ii) all tasks that depend on the lost events have been rolled back.

## 2.5 Rollback Restart for MPI Applications

Section 2.3 presents a wide variety of protocols. While these protocols are intellectually interesting, they are not very useful on their own. In order to become practical these protocols need to be embodied in an actual parallel computing system that is used by real users. Unfortunately, over the past three decades the theoretical work on rollback restart protocols has vastly outweighed practical research on bringing useful solutions to users. This section presents a novel non-blocking checkpoint coordination protocol and describes a practical implementation of this protocol for the MPI message passing API. This implementation is different from prior work on MPI rollback restart in that while prior work modifies existing open-source MPI implementations and is thus not applicable to the majority of real HPC

systems, the solution presented here works with any implementation of MPI.

### 2.5.1 The MPI Specification

The Message Passing Interface was first specified in 1994 with MPI 1.0 specification [3]. This specification was extended in 1995 to the 1.1 specification [4] and in 1997 to the 2.0 specification [1]. The basic motivation behind MPI is to provide a comprehensive cross-platform API for writing parallel message passing applications. It includes a wide variety of features, ranging from point-to-point communication, to complex datatypes, to collective communication and parallel file I/O. Over the past decade MPI has gained tremendously in popularity to a point where today it is a primary tool for writing parallel scientific applications.

There exist dozens of implementation of MPI, including both open-source and proprietary implementations. MPICH [101] [5] and MPICH2 [100] [6] are the reference MPI implementations targeted mostly towards clear structure and portability. MPICH has been extended by a number of different projects to produce optimized versions for different platforms or to experiment with adding certain functionality to MPI. MPICH-G2 [122] is a grid-enabled extension of MPICH. SCI-MPICH [215] is a port of MPICH optimized for clusters connected by Scalable Coherent Interface (SCI). MetaMPICH [173] extends MPICH to handle heterogeneous clusters. A number of projects, such as MPICH-V [47] extend MPICH with fault tolerance capabilities. LAM/MPI [195] is another popular open source implementation of MPI. It has also been extended with fault tolerance capabilities [187] and is being further extended via the OpenMPI project [90]. In addition to the open source implementations, there exist a number of proprietary implementations, including MPIPro [7], Quadrics MPI [8] and BlueGene/L MPI [32].

The MPI specification is quite large, providing applications with a wide variety of communication capabilities. An MPI application is made up of multiple processes that may run on the same or different processors. This section outlines the main features of the MPI 1.1 specification.

### 2.5.1.1 Point-to-point Communication

The basic primitive operation of any message passing system is sending a message to a given processor and receiving a message from some processor. Since each message has a single source and destination, these operations are known as point-to-point communication and are primarily supported in MPI via the `MPI_Send` and `MPI_Recv` calls.

`MPI_Recv` accepts a pointer to a data buffer, the id of the process from which the message is being expected, a numeric tag, a communication context and the address of an `MPI_Status` object. If the application does not care about the source of the message, it can use the special `MPI_ANY_SOURCE` process id. After the message is received and `MPI_Recv` returns, the `MPI_Status` object can be used to determine information about the incoming message, including its size and its sending process.

`MPI_Send` accepts a pointer to a data buffer, a numeric tag, a communication context and the id of the destination process and sends the contents of this buffer to that process. `MPI_Send` does not return until the send buffer may be reused by the application. It does not necessarily mean that the message itself has been received as the message may simply have been buffered by MPI. In order to provide the application with finer control over the communication, MPI provides three more specific variants of `MPI_Send`. `MPI_Bsend` ("buffered send") buffers the

message, meaning that it can return immediately after the message copy and is free to actually send the message across the network at its leisure. `MPI_Ssend` ("synchronous send") waits for the message to be received by the destination processor and does not return until this happens. `MPI_Rsend` ("ready send") is an optimization to be used when the application knows that the matching `MPI_Recv` call has been executed on the receiver process. The internal implementation of the regular `MPI_Send` is not specified beyond what is described above.

The tag and communication context arguments make it possible to establish multiple communication channels between pairs of processes. A given channel is identified by a specific tag-communication context combination and provides a guarantee of FIFO (first-in first-out) delivery for all messages sent on this channel. There are no FIFO guarantees for messages sent on different channels. While tags are simply integers, communication contexts are identified via `MPI_Comm` objects (discussed in Section 2.5.1.3) and can be manipulated by a number of specialized MPI functions.

`MPI_Send` and `MPI_Recv` provide a simple API for point-to-point communication but this API leaves limited opportunity for optimization. First, it is not possible for one process to be sending and receiving more than one message at the same time. Second, it is not possible for a process to be performing useful computation while waiting for a message to be sent or to arrive. As such, in addition to the "blocking" communication calls above, MPI includes "non-blocking" communication primitives.

`MPI_Irecv` is just like `MPI_Recv` except that it accepts a pointer to a `MPI_Request` object. `MPI_Irecv` returns immediately, before it is actually finished receiving the message. All the relevant information about this receive is stored in the

`MPI_Request` object. To determine whether this receive has completed, the application can apply functions such as `MPI_Wait` and `MPI_Test` to the `MPI_Request` object. A receive request is completed at exactly the same time as `MPI_Recv` would have returned. `MPI_Isend` related to `MPI_Send` just like `MPI_Irecv` relates to `MPI_Recv`, taking an extra `MPI_Request` object as an argument. There exist non-blocking variants of the three send specialization functions: `MPI_Ibsend`, `MPI_Irsend` and `MPI_Irsend`. `MPI_Wait` accepts an `MPI_Request` object and does not return until the corresponding send or receive has been completed. `MPI_Test` returns immediately and sets a flag that informs the application whether the request has completed or not. There exist variants of `MPI_Wait` and `MPI_Test` such as `MPI_Waitall` and `MPI_Testany` that apply these functions to multiple requests at the same time.

### 2.5.1.2 Collective Communication

While point-to-point communication gives the programmer all the power needed to implement efficient parallel applications, it does not make it easy. In reality there exist many common communication patterns that appear frequently in many applications. The MPI specification includes a number of such communication patterns as primitives in what is known as "collective communication". This has a positive effect on ease-of-use since complex communication patterns do not need to be coded up by hand every time they are needed. Furthermore, packaging up communication patterns as primitives allows MPI implementations to implement them in the most efficient way for a given system architecture. This allows for greater performance that is portable across different systems.

MPI collective routines can be broken up into three types: Single-sender, Single-

receiver, All-to-all and Barrier. A collective routine on a given process does not return until that process is finished participating in that operation. Each collective routine is defined for some subset of processors, identified in an `MPI_Comm` object that is passed as an argument. Communicators are discussed in Section 2.5.1.3 below.

### **Single-Sender Collectives**

These communication patterns have a single sender process that communicates data to one or more receivers. `MPI_Bcast` is a single-sender collective that implements the "broadcast" pattern. Each process that calls `MPI_Bcast` provides it with a pointer to a send/receive buffer and the id of the process that will be sending the data. MPI then delivers data from the sender process's buffer to the receiver processes' buffers. Another example is `MPI_Scatter` which is just like `MPI_Bcast`, except that the sender process's send buffer is broken up into a number of sub-buffers, one for each receiver process. MPI then delivers the contents of of a given receiver's send sub-buffer to that receiver's receive buffer.

### **Single-Receiver Collectives**

This category includes patterns where a single process is receiving data from multiple processes. `MPI_Gather` is a single-receiver collective that implements the inverse of `MPI_Scatter`. Every process calling `MPI_Gather` provides the address of a send/receive buffer and the id of the single receiver process. That process's receive buffer is broken up into multiple sub-buffers, one for each sender. MPI then delivers the data from each sender's send buffer to its respective receive sub-buffer at the receiver process. `MPI_Reduce` is a variant of `MPI_Gather` for cases where

the receiver wants to know some digest of the information held by other processes, rather than the information itself. In this case the receiver specifies a single receive buffer (rather than a collection of sub-buffers) and an associative and possibly commutative operation. MPI then reads the send buffers of all sender processes, applies the operation to their contents and writes the result to the receive buffer of the receiver process. For example, if the operation is  $+$  then `MPI_Reduce` will compute the sum of the numbers in all the send buffers and write this sum into the receive buffer.

### **All-to-All Collectives**

This is the most general category since it contains communication patterns where one or more processes send information to and receive information from one or more other processes. The primary example is `MPI_Alltoall`. When multiple processes call `MPI_Alltoall`, they specify both a send and a receive buffer. Each buffer is composed of multiple sub-buffers, one for each process that calls `MPI_Alltoall`. MPI then sends data from each send sub-buffer to its corresponding receive sub-buffer, allowing every process in the group to exchange information with every other process. `MPI_Allgather` and `MPI_Allreduce` are variants of `MPI_Gather` and `MPI_Reduce` where all the processes receive the result rather than just one process. `MPI_Reduce_scatter` performs a reduction of  $n$  numbers just like `MPI_Reduce`. It then breaks the resulting array of reduced numbers into  $n/p$  chunks (where  $p$  is the number of processes that called `MPI_Reduce_scatter` and scatters the chunks among all the participant processors, just like `MPI_Scatter`).

## Barrier Collectives

While the above communication patterns were defined in terms of how data flows from one process to another, the "barrier" pattern has no data semantics and only synchronization semantics. `MPI_Barrier` is the only example of barrier collectives. The semantics of `MPI_Barrier` are that when it is called by processes in a given group (identified by the `MPI_Comm` argument), no process in the communicator can return from this call until all the processes have also called it. No data is exchanged.

### 2.5.1.3 Opaque Objects

While the point of the MPI specification is to enable inter-process communication, a great deal of it focuses on a number of object types that make structured communication richer and easier. This includes structured datatypes for communication, communication groups and reduction operations, among others.

## Datatypes

In the above discussions, send and receive buffers were assumed to be unstructured regions of memory or number arrays that are exchanged between processes. While MPI can be used in this fashion, it provides much richer functionality for structuring the data that is communicated. This structure is provided via the `MPI_Datatype` object and its associated functions. MPI provides a number of default `MPI_Datatypes`, such as `MPI_BYTE` and `MPI_INTEGER`, including most of the basic types provided by the host language (currently Fortran and C/C++). It also provides a number of functions for hierarchically creating complex types out of these base types, including contiguous and non-contiguous arrays and structures

(similar to C's `structs`). This not only makes it easier to specify and communicate complex non-contiguous data structures but also allows MPI applications to run on heterogeneous hardware since it enables MPI to perform the appropriate format conversions during communication. (while officially legal this is rarely supported in real implementations)

All MPI communication routines accept a `MPI_Datatype` argument that describes the structure of the contents of the buffer. While this datatype can be trivially set to `MPI_BYTE`, if it is set to anything else the `MPI_Datatypes` specified by the sender and receiver processes must match each other in order for the communication to complete correctly.

## Groups and Communicators

While for many applications it is most natural to have a flat communication structure with a single global namespace for all the processes, it is frequently useful to partition processes into subgroups for various purposes. For example, a scalable simulation code may want to perform local broadcasts to inform the neighbors of a given process that its data has changed. This can be made easier by creating communicators that correspond to these local sub-groups and express the code's communication structure using these communicators.

MPI provides this grouping functionality via the `MPI_Group` ("group") and `MPI_Comm` ("communicator") objects. `MPI_Group` represents a set of processes. There are a few default groups like the empty group and the group of all processes and a number of functions to create new groups out of old groups and some filtering logic. Each group defines a ranking on its member processes, meaning that a given process may have a different id (called "rank") relative to each group that

it may belong to. To use a `MPI_Group` for communication the application needs to turn a `MPI_Group` into a `MPI_Comm`. In addition to grouping processes, MPI communicators provide a communication context in the sense that communications happening using different `MPI_Comm` objects have no ordering guarantees relative to each other, allowing for higher performance. When using a given communicator to communicate with other processes, it is necessary to use their ranks within the communicator's process group rather than in the flat global namespace.

In order to provide more convenience for applications with standardized communication structures MPI supports the notion of "topologies". A topology is a way to take a group of processes that can be conveniently embedded into a cartesian grid or a graph and allow the application to refer to members of this group using this logical structure instead of using the regular ids.

### **User-defined Reduction Operations**

While functions such as `MPI_Reduce`, and `MPI_Allreduce` provide the application with a number of default reduction operations like addition, product, max and min, it is frequently useful for the application to define its own operations. MPI provides this functionality via the `MPI_Op` object. An `MPI_Op` is created by the `MPI_Op_create` function that takes in an associative and possibly commutative user-defined function and produces a `MPI_Op` object to represent it. This `MPI_Op` can be passed into `MPI_Reduce`, `MPI_Allreduce`, etc. as a regular reduction operation, causing the specified user function to be called by MPI for the purpose of performing the reduction.

## 2.5.2 Prior Work on Rollback Restart

The popularity of MPI in the HPC community has motivated multiple groups to develop rollback restart systems for MPI applications.

### 2.5.2.1 Manual Solutions

The simplest way of doing this is to punt on the question and assume that the application will deal with MPI on its own, reinitializing the MPI library, implementing its own restart protocol for recreating communication and opaque MPI objects such as datatypes and communicators. This can be considered as "default support" by checkpointers that do not explicitly deal with MPI but can checkpoint applications that happen to use MPI.

A more user-friendly solution is provided by FT-MPI [82] [81]. While MPI does not have clean semantics for how implementations must deal with failures, FT-MPI provides its own much more robust specification. The most important service provided by FT-MPI is to allow applications to constructively deal with failures. When a processor failure is detected FT-MPI allows the application to

- abort the computation;
- initiate a restart procedure (described below);
- do nothing and produce an error every time the application tries to communicate with a failed process.

FT-MPI provides specific semantics for what happens to MPI opaque objects on the surviving processes. Any objects with purely local information (datatypes, reduction operations, etc.) work as before with no modification. Objects with

non-local information (groups and communicators) are destroyed. If the user wants FT-MPI to perform restart, these objects are recreated in one of three ways:

- `FTMPI_COMM_MODE_REBUILD`: failed processes are replaced by new processes, which take their place in any groups and communicators on other processes that previously contained them
- `FTMPI_COMM_MODE_BLANK`: the spots occupied by the failed processes are left blank in their former groups and communicators
- `FTMPI_COMM_MODE_SHRINK`: any groups and communicators that used to contain the failed processors are shrunk so as not to contain them any more, causing ranks of processes within these groups and communicators to change

In the process of recovering from a failure it is important for the application to get clear semantics about messages that were in-flight at the time of the failure. FT-MPI provides two modes of operation:

- cancel all in-flight messages, and
- complete all in-flight messages, only cancelling messages to and from failed processes.

While FT-MPI does not on its own provide an automated fault tolerance solution for MPI applications, it is an important aid. In particular, [61] shows how an application running on top of FT-MPI can be made fault tolerant by reliably saving the checkpoint of  $n$  processes on  $n+m$  processes using a specially developed error correcting code.

### 2.5.2.2 Automated Implementation-specific Solutions

Given the significant amount of manual work required by the above approaches, many projects have tried to make rollback restart of MPI applications easy by automatically recreating MPI state. Given the complexity of the MPI specification, these projects have made the job more tractable by taking an existing implementation of MPI and modifying it internally to give it rollback restart functionality.

MPICH-V [49] [47] [48] is an extension of MPICH that implements a number of rollback restart protocols. MPICH-V1 [47] is a form of pessimistic message logging that routes all messages through "channel memory" processors that perform all the logging. MPICH-V2 [48] is a more pessimistic message logging protocol where senders log outgoing messages. [49] presents MPICH-V/causal and MPICH-V/CL, which are implementations of causal message logging and the Chandy-Lamport coordinated checkpointing protocol, respectively, developed on top of the MPICH-V platform. It also presents results of experiments that compare the performance of MPICH-V2, MPICH-V/causal and MPICH-V/CL.

The implementation of MPI used on the BlueGene/L supercomputer [27] can checkpoint the state of an MPI application, including the internal state of the MPI implementation. The major limitation is that they assume that there are no outstanding messages during a checkpoint.

[187] and [218] discuss the independent efforts by two teams to integrate LAM/MPI [195] and the BLCR [75] sequential checkpointer into a checkpointer for MPI applications. Both schemes rely on LAM's native ability to integrate with sequential checkpointers. In order to perform a checkpoint LAM first flushes all communication channels and then tells the sequential checkpointer to checkpoint each MPI process (making it a variant of the sync-and-stop protocol discussed in

Section 2.3.4.1). Since all internal MPI state is checkpointed along with the state of each process, on restart LAM has enough information to recreate its processes and resume communication. The OpenMPI [90] project extends LAM/MPI and a few other MPI implementations in a number of ways, including a new framework for integrating more complex rollback restart protocols into OpenMPI.

SMPCkpt [72] is an extension of MPICH that is specifically aimed for symmetric-multiprocessor machines. In addition to providing a specialized MPI implementation SMPCkpt can provide checkpoint-restart for MPI applications by integrating itself with the libckpt [169] sequential checkpointer. It implements both a blocking coordination protocol and non-blocking checkpoint protocol that is similar to Chandy-Lamport.

Starfish [29] is a communication system designed for manageability, dynamic process management and high availability. It is built on top of the Ensemble [206] group communication system. Although Starfish provides additional functionality beyond that provided by MPI, it can be used directly by unmodified MPI-2 programs. Since Starfish is a general communication system that is designed for reconfigurability, it can in principle support any rollback restart protocol. [29] reports on their implementation of the sync-and-stop checkpointing protocol.

BCS-MPI [84] is an implementation of MPI based on Buffered CoScheduling [88]. BCS-MPI divides the application's execution into 500ns time slices, each of which is terminated by a global synchronization. During each time slice BCS-MPI schedules all communication for the next time slice. Because communication is only happening during time slices and not across them, it is possible to checkpoint the state of the entire application during such a synchronization without having to worry about in-flight communication. This structure makes it

easy to checkpoint this implementation of MPI [163] since all processes will be checkpointed at the same time.

AMPI [113] [112] is an implementation of MPI built on top of the Charm++ [121], a system that runs large numbers of objects (called "chares") in parallel on a smaller number of processors, providing a message passing communication interface. [219] presents FTC-Charm++, a checkpoint-restart extension to Charm++ that uses the idea of processor virtualization to provide fast restarts. When a processor goes down, its chares are restored on either a spare processor (the trivial case) or one of the remaining processors. In the latter case Charm++'s load balancing infrastructure can rebalance the allocation of chares to processors, returning the computation back to balance and high performance. Furthermore, instead of using a centralized file system, FTC-Charm++ improves performance by storing each processor's checkpoint in the disks or memories of other processors, making copies to improve reliability. Since AMPI is built on top of Charm++, it gets these performance benefits as well.

Zap [160] is kernel-level migration system for Linux. Unlike most conventional checkpointers that focus on individual processes, Zap groups processes into "pods" and provides migration at the pod level. While processes are allowed to interact freely with other processes within the same pod, their interactions with processes outside their pod are severely restricted in order to prevent dependencies to processes that will not be present after a pod migrates. Intra-pod interactions are mediated by a virtualization layer that provides processes with a private namespace, a private view of the file system and a migratable network subsystem, among other things. The network subsystem is limited to TCP/IP communication and provides each pod with its own IP address. While this address may change

as the pod is migrated, dynamic DNS and other transport-level techniques are used to ensure that all communication destined for pod processes gets delivered to the pod's new location and that the pod processes are not aware of the migration (unless they need to be aware). While Zap does not directly address MPI communication, many MPI libraries can use TCP/IP to carry their communication. While this is generally not a high-performance solution, it is adequate in many case such as loosely coupled applications running on networks of workstations.

Migration can also be done below the operating system, as shown in [63], which uses the Xen virtual machine manager [64] to migrate the state of the system software stack. This includes any applications and libraries (including the MPI implementation, if any) along with the entire state of the operating system. While migrating the software state becomes easy, the difficult point becomes the migration of device state since devices such as disks and internet connections are external to the virtual machine. The Xen solution to the migration of network connections is to maintain a static IP address for each OS image, even if it is migrated, using a number of different techniques (e.g. unsolicited ARP replies) to inform other machines on the network of the migration. (these techniques apply to many but not all real-world network environments). Because of the IP address persistence requirement, the current solution does not allow for migration across networks.

### **2.5.2.3 Automated Implementation-independent Solutions**

While the above systems do a good job of providing fault tolerance to users of their respective implementations of MPI and/or supported operating systems and hardware, they have a significant weakness in that no given solution is applicable to the majority of real HPC systems. The weakness of the MPI implementation-level so-

lutions is that most of these rollback restart-enabled MPI implementations are not used in practice. The problem for lower level solutions such as Zap and Xen is that they focus exclusively on TCP/IP and do not support high-performance communication libraries. Real high-performance systems use specialized implementations of MPI that have direct access to the networking hardware to give users the maximum amount of performance obtainable on a given system. Meanwhile, the above systems, while they do provide a useful and needed service, remain as research prototypes that have not seen wide acceptance due to their lack of performance and the kind of dedicated development support required from a industrial-quality MPI implementation. The only exception is OpenMPI, which is actively being developed for cross-platform performance as well as rollback restart but even OpenMPI has yet to see wide acceptance.

The fundamental problem is that with at least a dozen high performance MPI implementations in active use today, focusing on TCP/IP or enhancing a single implementation of MPI with rollback restart capabilities does not qualify as anything more than a research exercise: useful for evaluating protocol ideas but not robust enough or sufficiently widely available to become useful to real users.

Figure 2.22 presents the typical system stack for MPI applications and an analysis of the implementation complexity/generality tradeoffs inherent in implementing rollback restart for MPI applications. It shows an application running on top of the MPI specification. This specification is implemented by one of many libraries. These libraries run on top of network interfaces, which are implemented by actual network subsystems. In this layered model rollback restart can be provided by embedding rollback restart functionality into any level. Most prior work focuses on modifying the MPI implementation and/or the application, although

System Stack	RR Solution	Implementation Complexity	System Count	Implementation Effort
		Medium	Very High	High
		Medium - High	Very Low	Low - Medium
		Low - Medium	Very High	Medium - High
		Medium	Medium - High	Medium - High
		High	Very High	High - Very High

Figure 2.22: Space of MPI checkpointing solutions

other options are possible.

The tradeoff between the complexity and generality of providing rollback restart for the vast majority of real applications and machines will be evaluated by using an "implementation effort" metric. This metric depends on two factors:

- Implementation Complexity:** The complexity of implementing solutions at this level, including any performance challenges that need to be overcome. For example, at the network interface level, this is the average complexity of network interface APIs.
- System Count:** The number of different systems that exist at this level, most of which need to be augmented with rollback restart capability in order to make it available to most users. This measure is increased if there is no

public access to the system source code needed to do the implementation. At the network interface level, this is the number of different network interface APIs that may need to be supported, with systems like Quadrics presenting extra difficulty because they do not make their source code or driver specification available to others.

In addition to the system stack, Figure 2.22 shows the *implementation complexity*, *system count* and *implementation effort* of implementing rollback restart at each possible level. These alternative and their respective scores are discussed below.

We can provide rollback restart at the application level by modifying applications themselves to coordinate their own state-saving. This is done frequently in practice and can be made quite efficient via techniques such as Algorithm-Based Fault Tolerance [114] [37] and the programmer's knowledge about the application's communication pattern (which may be BSP or master-slave). Although it is difficult to estimate *implementation complexity* involved in adding rollback restart to the "average" application, it can be considered to be Medium. The reason is that depending on the details of the application and the scalability required of the rollback restart protocol, *implementation complexity* can range from very low to very high. One big problem with modifying the application is that the *system count* of this approach is huge, with each new application presenting a different system that needs to be modified for rollback restart. Thus, the *implementation effort* of modifying the application can be considered High.

The next level down is the MPI specification. It is possible to implement rollback restart at this level by creating a thin layer between the application and the MPI implementation. This layer would present MPI semantics to the application and would run on top of MPI but would also implement some rollback restart

protocol. As such, unlike regular MPI, it would continue to provide the application with MPI semantics even if some processes have been rolled back. As such, the primary challenge in developing such a solution is to checkpoint and recreate the application-visible state of the MPI implementation while only using functionality provided by the MPI specification itself. Because of the size of the MPI spec, the *implementation complexity* of this approach is Medium to High. At the same time, the fact that MPI changes very slowly (the last version of the specification came out in 1997) means that the *system count* of this solution is Very Low. Thus, *implementation effort* of doing rollback restart at the MPI specification level is Low to Medium. There is no prior work at this level.

Moving down the system stack are MPI implementations. All of the prior work described in Section 2.5.2 works at this level: modifying specific implementations of MPI to enable rollback restart. The strength of this approach is the Low to Medium *implementation complexity* since the internal details of a given MPI implementation are well-known to its designers and can be abstracted into a fairly compact internal message passing interface. However, the large number of different MPI implementations, many of which are proprietary (the biggest problem) causes the *system count* to be Very High. This makes the overall *implementation effort* of modifying MPI implementations to enable rollback restart to be Medium to High.

The next level is the network interface, which is the set of APIs that connect different network subsystems to higher level communication systems such as implementations of MPI and sockets. While these APIs tend to be much less complex than MPI, virtualizing them can have an undesirable performance cost since all network characteristics that may change on restart need to be virtualized. Thus, *implementation complexity* of this approach is Medium. While the number of dif-

ferent network interface APIs is not as large as number of network vendors, it is still a fairly large number. Furthermore, as interfaces change over time and new interfaces come into use, the effective *system count* is Medium to High. Thus, the final *implementation effort* of performing rollback restart at the virtual network interface level is Medium.

The lowest level is the network subsystem hardware. It is possible to modify this hardware to implement one or more rollback restart protocols, as done in the IBM SP-2, [194] and [175]. While the variety of hardware designs makes it difficult to estimate the *implementation complexity* of modifying them to implement rollback restart protocols, the resource and design limitations of hardware make this a generally complex task. Thus, the *implementation complexity* is High. Furthermore, this same variety of hardware designs and implementations makes the *system count* of this approach Very High, resulting in a High to Very High *implementation effort*.

In addition to *implementation complexity* another important property of the above approaches to parallel rollback restart is *checkpoint flexibility*. This is the set of times when each process may record its checkpoint. For any solution level, it is not possible to record the checkpoint of the process while in the middle of a call to the lower levels. For example, if rollback restart is implemented inside the MPI implementation, it is not possible to checkpoint in the middle of a call to the underlying network interface. Similarly, a solution that works above the MPI implementation cannot checkpoint the process while the process is in the middle of an MPI call. The reason is simple: if control is currently inside some lower-level functionality, high-level state of the system is temporarily undefined. As such, a checkpoint of such state can be erroneous. *Checkpoint flexibility* is

certainly a relevant metric, with applications like system fault tolerance requiring little flexibility while applications like local rollback for processor mis-speculation purposes requiring cycle-level *checkpoint flexibility*. However, given the general unavailability of rollback restart for most real-world systems, this metric is secondary to *implementation complexity* (since optimizing the *checkpoint flexibility* of nonexistent rollback restart systems is pointless) and is thus not be considered here.

Given the different tradeoffs involved in implementing rollback restart in a real system, it becomes apparent that implementing it as a layer between the application and the MPI implementation presents us with the lowest *implementation effort*. This is because the complexity of dealing with the large MPI specification is balanced by the fact that there is only one such specification and it changes very slowly. The challenges of working at this level are two-fold:

- While checkpointing solutions at lower levels know a significant amount of information about the low-level details of the communication infrastructure, a solution that works at the MPI specification level knows only the information that can be obtained via this interface. Since MPI is meant to provide cross-platform communication functionality, it provides little information about low-level details. As such, it is not clear if such a high-level solution can provide efficient rollback restart.
- MPI is a large and complex API. Solutions that work below this level can avoid this complexity but a solution at the MPI specification level has to deal with it directly since it needs to save and restore all the application-visible MPI state while using only the MPI API. It is not clear that this is achievable.

While the challenges are significant, so are the benefits:

- If rollback restart can be efficiently implemented at the MPI specification level, then the problem of rollback restart for MPI applications (a large fraction of all parallel applications) will be solved, with updates only necessary when the MPI specification itself changes (a rare occurrence).
- A rollback restart system that works across different platforms and MPI implementations makes it possible to experimentally evaluate and compare a variety of protocols and checkpointing techniques relative to many real-world MPI applications and real computing clusters, using the most efficient communication infrastructure on each cluster.

With these benefits and challenges in mind, the MPI specification-level approach was taken in the RRAMP(Rollback Restart for MPI) system, described in the rest of this Section.

### 2.5.3 RRAMP

RRAMP (Rollback Restart for MPI) is system that provides rollback restart for MPI applications that may run on any implementation of MPI. It is implemented as a thin layer between the application and the MPI library (as shown in Figure 2.23) and uses only the functionality available from the MPI specification to provide the application with semantics identical to MPI 1.1 even in the presence of rollbacks. As such, it uses the Replay methodology from the 4R classification. Since the focus of RRAMP is on the MPI specification, it assumes that the rest of the application and any of the other libraries it uses are checkpointed via some other means (e.g. using a sequential checkpointer such as BLCR [75], licsm [218],

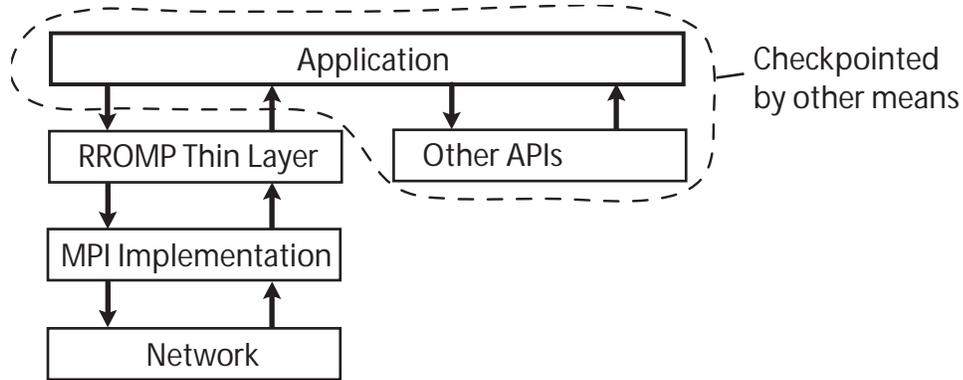


Figure 2.23: Architecture diagram of RRAMP

TICK [96], libckpt [169], ckpt [9], epckpt [10], UCLiK [87], CHPOX [159], CryoPID [11], NT-SwiFT [135], [25], [196], Winckp [62], WinNTCkpt [217], [146], among others).

Since the function of RRAMP is to provide the same semantics as MPI, together with additional reliability guarantees, the following sections will go through the various portions of the MPI specification, describing how they are handled by RRAMP. The current implementation of RRAMP includes a non-blocking coordinated checkpointing protocol (Section 2.3.4.2) that generates criss-crossed restart lines. The primary contributions of RRAMP are:

1. it is the first rollback restart system that operates at the MPI specification level, allowing it to take advantage of the relatively low *implementation effort* of solutions at this level, and
2. the protocol used by RRAMP is the first checkpointing protocol that generates criss-crossed restart lines. As discussed in Section 2.2.1, this makes it compatible with both sequential checkpointers that can checkpoint at any point in time and those that may only checkpoint at special locations in the application's source code.

## 2.5.4 Checkpointing Protocol

As discussed in Section 2.2, a criss-crossed restart line is one that is crossed by both late and early messages, as shown in Figure 2.24. Recall that checkpointing protocols are defined as protocols that generate restart lines exclusively composed of prior checkpoints, rather than current process states. Section 2.1 describes the issues presented by past, future, late and early messages and what must be done in order to deal with them. In short, to successfully restart an application from such a restart line we must do the following.

1. Roll each process back to its respective local checkpoint.
2. Resume the execution of each process.
3. On restart, when the application tries to send an early message (by calling an MPI send function in the RRAMP layer), this request is not passed on to MPI.
4. On restart, when an application tries to receive a late message (by calling an MPI receive function in the RRAMP layer), the receive request is not passed on to MPI and the data of this message is copied from the checkpoint directly to the application's message receive buffer.
5. On restart, when the application performs a non-deterministic operation that precedes the sending of some early message, this operation is forced to happen exactly the same way as it did during the original execution.

RRAMP's checkpointing protocol ensures that the above conditions are satisfied and only assumes that (i) if no failure is detected, the delivery of messages is reliable and (ii) all messages sent by each process must eventually be received. In

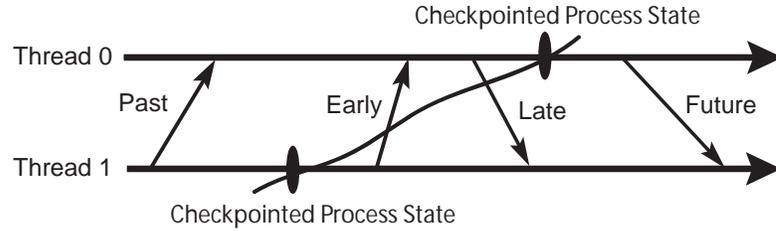


Figure 2.24: Restart line made up of checkpointed process states and crossed by past, future, late and early messages

particular, no FIFO delivery assumption is made, as was the case with Chandy-Lamport. Each process may record a local checkpoint at any time it feels convenient. This may be based on a timer or because the application has reached a potential checkpoint location inside the source code. While the protocol works correctly regardless of the relative times of the local checkpoints on different processes, it is most efficient when these checkpoints are taken roughly at the same time.

#### 2.5.4.1 High-level description of protocol

There is a single *leader* process whose task it is to coordinate checkpointing, as described below. It can be chosen via any leader election algorithm. In RRAMP, the task with rank 0 in the `MPI_COMM_WORLD` communicator is the leader.

**Phase #1** When a some process  $p$  decides to take its  $i^{th}$  local checkpoint ( $C_{p,i}$ ), it saves its local state and the identities of any received early messages on stable storage. It then starts writing a log of (i) every late message it receives, and (ii) the result of every non-deterministic decision it makes. It then enters checkpoint interval  $I_{p,i}$ . Once a process has received all of its late messages, it sends a control message called *readyToStopLogging* to the leader, but continues to write non-deterministic decisions to the log.

**Phase #2** When the leader receives a *readyToStopLogging* message from all

processes, it knows that every process has taken its local checkpoint. Since every process has transitioned to checkpoint interval  $I_{*,i}$ , any message sent by any processor after the leader has acquired this knowledge cannot be an early message. Therefore, all processes can stop logging non-deterministic events. To share this information with the other processes, the leader sends a control message called *stopLogging* to all other processes.

**Phase #3** An application process stops logging non-deterministic events and data of late messages when (i) it receives a *stopLogging* message from the leader, or (ii) it receives a message from a process that has itself stopped logging. (note that it must have stopped logging incoming late messages before it can receive a *stopLogging* message)

The second condition is a little subtle. Because no assumptions are made about message delivery order, it is possible for the following sequence of events to happen.

1. Process  $p$  receives a *stopLogging* message from the initiator, and stops logging.
2.  $p$  makes a non-deterministic decision.
3.  $p$  sends a message containing this decision to process  $q$ , which is still logging.
4. Process  $q$  uses this information to create an event that it logs.

When  $q$  saves its log, we have a problem: the saved state of the global computation is causally dependent on an event that was not itself saved. To avoid this problem, we require a process to stop logging if it receives a message from a process that has itself stopped logging. These conditions for terminating logging can be described quite intuitively as follows: a process stops logging when it hears from the leader or from another process that all processes have taken their checkpoints.

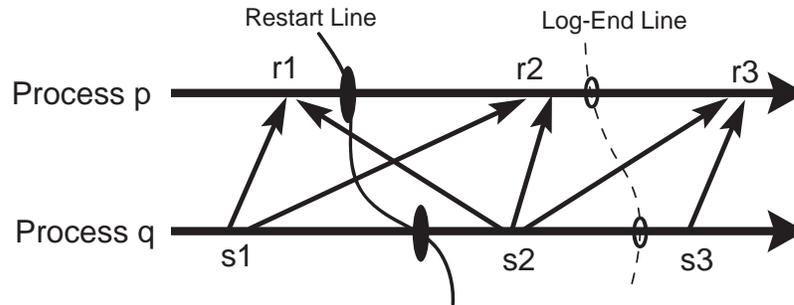


Figure 2.25: Possible types of messages that may happen in protocol

**Phase #4** Once the process has saved its log on disk, it sends a *stoppedLogging* message back to the initiator. When the initiator receives a *stoppedLogging* message from all processes, it commits the checkpoint on stable storage, and terminates the protocol.

The protocol provides the following guarantees:

**Claim 1**

1. No process can stop logging until all processes are beyond the restart line.
2. A process  $p$  that is beyond the log-end line cannot send a message to a process  $q$  that is behind the log-end line.
3. A process  $p$  that is beyond the log-end line cannot receive a message from a process  $q$  that is behind the restart line.

Figure 2.25 shows the possible communication patterns, given these guarantees. For example, a message sent by process  $q$  at point  $s1$  (behind the restart line) cannot be received by process  $p$  at point  $r3$  (beyond the log-end line).

#### 2.5.4.2 Piggybacked information on messages

To implement this protocol, the protocol layer must piggyback a small amount of information on each application message. The receiver of a message uses this

piggybacked information to answer the following questions.

- Is the message late, early or intra-interval (past or future)?
- Has the sending process stopped logging?
- Which messages should not be resent during restart?

The piggybacked values on a message are derived from the following values maintained on each process by the protocol layer:

- *intervalNum*: This integer keeps track of the checkpoint interval in which the process is. It is initialized to 0 at start of execution, and incremented whenever that process takes a local checkpoint.
- *amLogging*: This is a boolean that is true when the process is logging non-deterministic events and the data of late messages, and false otherwise.
- *nextMessageID*: This is an integer which is initialized to 0 at the beginning of each interval, and is incremented whenever the process sends a message. Piggybacking this value on each application message ensures that each message sent by a given process in a particular checkpoint interval has a unique ID.

A simple implementation of the protocol can piggyback all three values on each message that is sent by the application. When a message is received, the protocol layer at the receiver examines the piggybacked interval number and compares it with the interval number of the receiver to determine if the message is late, early or intra-interval. By looking at the piggybacked boolean, it determines whether the sender is still logging. Finally, if the message is an early message, the receiver logs

the pair  $\langle sender, messageID \rangle$ . These pairs are saved to stable storage when the processor takes its local checkpoint. During restart, these pairs are retrieved from stable storage by the receivers of these messages, and the senders of these early messages are informed of the messageIDs so that on restart when the application tries to resend these messages, these sends can be suppressed.

Further economy in piggybacking can be achieved if we exploit the fact that at most one global checkpoint can be ongoing at any time and that all the checkpoints that make up a given global restart line must have the same checkpoint number  $C_{*,i}$ . This means that the intervals of processes can differ by at most one. Let us imagine that intervals are colored red and green alternatively. When the receiver is in a green interval, and it receives a message from a sender in a green interval, that message must be an intra-interval message (i.e. past or future relative to a given restart line). If the message is from a sender in a red interval, the message could be either a late message or an early message. It is easy to see that if the receiver is not logging, the message must be an early message; otherwise, it is a late message. Therefore, a process need only keep track of the color of its interval, and this color can be piggybacked instead of the interval number. With this optimization, the piggybacked information reduces to two booleans and an integer.

Further optimization is possible. If 32-bit integers are used, the two most significant bits of an integer can be used to represent the color of the interval and the state of the *amLogging* flag of the sender, and remaining 30 bits can be used as the *messageID*. This solution should work fine because it is unlikely that a single process will send more than a billion messages between checkpoints! With this optimization, the protocol can be implemented by piggybacking a single integer on the application payload.

### 2.5.4.3 Completion of receipt of late messages

Finally, we need a mechanism for allowing an application process in one interval to determine when it has received all the late messages sent in the previous interval. Protocols such as the Chandy-Lamport algorithm assume FIFO communication between processes, so they do not need explicit mechanisms to solve this problem. Since we do not assume FIFO communication, we need to address this problem.

The solution we have implemented is straight-forward. In every interval, each process  $p$  remembers how many messages it sent to every other process  $q$  (call this value  $sendCount_{p \rightarrow q}$ ) during this interval. Each process  $p$  also remembers how many messages it received from each other process  $q$  that were sent by  $q$  during the last interval (call this value  $lastRecvCount_{p \leftarrow q}$ ). When a process  $q$  takes its local checkpoint, it sends a *mySendCount* message to every other process  $r$ , which contains  $sendCount_{q \rightarrow r}$ , the number of messages  $q$  sent to  $r$  in the previous interval. When process  $p$  receives this control message, it can compare the received  $sendCount_{q \rightarrow p}$  with  $lastRecvCount_{p \leftarrow q}$  to determine how many more messages to wait for before it is sure that no more late messages may come from  $q$ .

A minor detail is that a process  $P$  actually needs to keep *two* receive counts for each process  $Q$  that may send it messages; this is because late messages from  $q$  to  $p$  sent in one interval may be interspersed with intra-interval messages from  $p$  to  $q$  sent in the next interval. In the protocol given below, these two counters are called *lastRecvCount* and *currentRecvCount*.

A more subtle issue is the following: since the value of  $sendCount_{p \rightarrow q}$  is itself sent in a control message, how does  $q$  know how many of these control messages it should wait for? A simple solution is to assume that every process may communicate with every other process in every interval, so a process expects to receive a

*sendCount* control message from every other process in the system. This solution works, but if the topology of the inter-process communication graphs is sparse, most *sendCount* control messages will contain 0, which is wasteful. If the topology of this communication graph is sparse and fixed, we can set up a data structure in the protocol layer that holds this information. There are fancier solutions for the case when the communication topology is sparse and dynamic such as the coordination protocols in [185]. In the pseudo-code of Figure 2.26, we assume that the inter-process communication graph is fixed, and we use the terms *senders* and *receivers* to denote the set of processes that send messages to a given process, and the set of processes that are sent messages by a given process respectively.

#### 2.5.4.4 Putting it all together

Figure 2.26 is a synthesis of the mechanisms discussed above into a single protocol which is executed by the protocol layer at each processor,  $p$ .

Each process  $p$  maintains the following variables:

- *intervalNum<sub>p</sub>*: the process's current checkpoint interval number.
- *amLogging<sub>p</sub>*: flag indicating whether the process is currently logging the data of late messages and the outcomes of non-deterministic events.
- *nextMessageID<sub>p</sub>*: the id of the next message sent. Initialized to 0 and incremented every time a message is sent by process  $p$ .
- *sendCount<sub>p→q</sub>*: the number of messages sent by process  $p$  to process  $q$  in the current checkpoint interval. Initialized to 0.
- *sentFromLastCount<sub>p←q</sub>*: the number of messages sent by process  $q$  to process  $p$  in the last checkpoint interval. Initialized to 0.

- *currentRecvCount<sub>p←q</sub>*: the number of messages received by process  $p$  from process  $q$  that  $q$  sent in the current checkpoint interval (these messages must have been intra-interval). Initialized to 0.
- *lastRecvCount<sub>p←q</sub>*: the number of messages received by process  $p$  from process  $q$  that  $q$  sent in the last checkpoint interval (these messages must have been late relative to the last restart line or intra-interval inside the last checkpoint interval). Initialized to 0.
- *earlyIDs<sub>p←q</sub>*: list of IDs of early messages received by processor  $p$  from processor  $q$ . Initialized to nil.

The following control messages are exchanged during the protocol:

- *mySendCount*: Sent from process  $q$  to process  $p$  to inform  $p$  of the number of messages  $q$  sent to  $p$  in the checkpoint interval that just completed.
- *readyToStopLogging*: Sent by a processor to the leader when it has taken its local checkpoint and it has received incoming all late messages.
- *stopLogging*: Sent from the leader to other processors when it has proof that all processors are ready to stop logging.
- *stoppedLogging*: Sent by a processor to the leader when it has stopped logging.

#### 2.5.4.5 Restart

On restart all processes read their respective checkpoints and exchange their *earlyIDs*, with each process  $p$  sending *earlyIDs<sub>p←s</sub>* to process  $s$ . They then resume their

Pseudo-code for process  $p$ .

`communicationEventHandler()`

Application message send to process  $q$ :

Piggyback  $\langle intervalNum_p, amLogging_p, nextMessageID_p \rangle$  on the message  
 $sendCount_{p \rightarrow q} ++$   
 $nextMessageID_p ++$

Application message receive from process  $q$ :

Remove  $\langle intervalNum_q, amLogging_q, messageID_q \rangle$  from the message

early message: // assert  $amLogging_q$  and not  $amLogging_p$

append  $messageID_u$  to  $earlyIDs_{p \leftarrow q}$

intra-epoch message:

if ( $amLogging_p$  and not  $amLogging_u$ )

`finalizeLog()`

$currentRecvCount_{p \leftarrow q} ++$

late message: // assert  $amLogging_p$  and not  $amLogging_q$

append message to log

$lastRecvCount_{p \leftarrow q} ++$

`receivedAll?()`

Control message: *stopLogging*

`finalizeLog()`

Control message: *mySendCount(n)* from process  $q$

$sentFromLastCount_{p \leftarrow q} \leftarrow n$

if ( $amLogging_p$ ) //  $p$  has taken its own checkpoint

`receivedAll?()`

Figure 2.26: Checkpointing protocol pseudo-code

Figure 2.26 (Continued)

```

receivedAll?()
    if (for all senders  $s$   $lastRecvCount_{p \leftarrow s} = sentFromLastCount_{p \leftarrow s}$ )
        send readyToStopLogging message to leader
         $sentFromLastCount_{p \leftarrow s} \leftarrow \infty$  for all senders  $s$ 
finalizeLog()
    write log to stable storage
     $amLogging_p \leftarrow false$ 
    send stoppedLogging message to leader
takeCheckpoint()
    save node state to stable storage
     $intervalNum_p ++$ 
    for each receiver  $r$ 
        send  $mySentCount(sendCount[sendCount_{p \rightarrow r}])$  to  $r$ 
    for each sender  $s$ 
         $lastRecvCount_{p \leftarrow s} = currentRecvCount_{p \leftarrow s}$ 
         $currentRecvCount_{p \leftarrow s} = earlyIDs_{p \leftarrow s}$ 
        save  $earlyIDs_{p \leftarrow s}$  to stable storage
         $earlyIDs_{p \leftarrow s} \leftarrow nil$ 
     $amLogging_p \leftarrow true$ 
     $nextMessageID_p \leftarrow 0$ 
receivedAll?()

```

execution in a special managed mode, during which each process executes deterministically, suppresses sends of early messages and replays the data of late messages, using data from the log and the received *earlyIDs* lists.

When the process tries to perform a non-deterministic operation, the outcome of this operation is dequeued from the front of the log and it is repeated exactly as recorded. When a process  $s$  tries to send a message to process  $p$ , it checks the *earlyIDs* <sub>$p \leftarrow s$</sub>  list that it received from  $p$  to see if the message's ID is in the list and if so, this entry is removed and the send operation is not passed on to the underlying communication system. Similarly, if a process  $p$  tries to receive a message, it examines to front of the log to see if this entry contains the data corresponding to its message ID. If so, it is dequeued from the log, its data is copied to the message receive buffer and the receive operation is not passed on to the underlying communication system.

Since restart execution is deterministic, the application is guaranteed to perform exactly the same sequence of non-deterministic operations, sends and receives (with the same message IDs) on restart, at least upto the time it sent its last early message. Each process exits its managed execution when the log and all of its *earlyIDs* lists are empty.

### 2.5.5 Blocking Point-to-Point Communication

MPI blocking point to point communication routines include `MPI_Recv` and the four variants of `MPI_Send`. As these semantics of these routines are either equivalent to or stronger than the semantics of the of sends and receives in the abstract model, the above protocol applies to them directly. One notable point is that in MPI interface level, a message only arrives when the application has posted a

receive for it. Indeed, it does not matter if the message has actually arrived at the local network card or the processor itself until this information has propagated through the MPI interface. Since the protocol above does not commit a given global checkpoint until all late messages crossing its restart line are received, we need to make the assumption that the application receives all of the messages that it sends in a relatively short amount of time.

## 2.5.6 Non-blocking Point-to-Point Communication

Non-blocking point-to-point communication is different from its blocking counterpart because the act of starting a send or receive operation is decoupled from the act of learning that it has completed. In MPI this means that a blocking `MPI_Recv` call must be broken into an `MPI_Irecv` that initiates a receive and `MPI_Wait/MPI_Test` to determine when the message has actually arrived. `MPI_Send` and its variants are broken up in the same fashion.

### 2.5.6.1 Applying the Protocol

To apply the above protocol to non-blocking point-to-point communication we need to focus on the meaning of a message send or receive in the above model. A message send is the point in time when the application has told the underlying communication system to issue this message and from that point need do nothing else to ensure correct delivery. A message receive is the point in time when the application has received the data of the message. Given these semantics it becomes clear that the message send event of the abstract message passing model corresponds directly to `MPI_Isend` and its variants. While `MPI_Wait/MPI_Test` tell the application when it is safe to reuse the message buffer, it is the original call

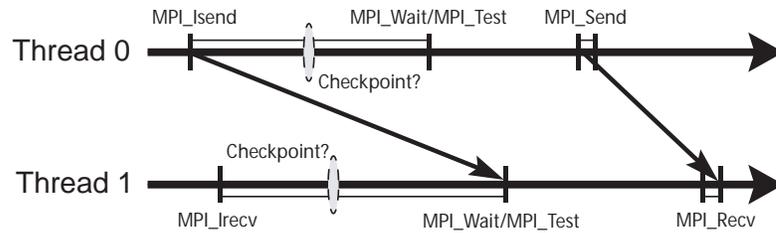


Figure 2.27: Representation of blocking and non-blocking communication in abstract model

to `MPI_Isend` that initiates the send itself. However, the message receive event of the abstract model corresponds not to the `MPI_Irecv` call but to the point in time when `MPI_Wait/MPI_Test` tell the application when the message has actually arrived. This is clear because while `MPI_Irecv` is simply a way for the application to inform MPI about the location of the receive buffer, it is only when `MPI_Wait/MPI_Test` return successfully that the application knows that the message has been received.

Figure 2.27 shows the difference between how blocking and non-blocking point to point messages are represented in the abstract message passing model. Note that although it is not possible for the application to take a checkpoint in the middle of a `MPI_Send` or `MPI_Recv` call, it is easily possible for the application to checkpoint in the middle of a non-blocking send or receive, that is, between the `MPI_Send` or `MPI_Recv` call and the corresponding `MPI_Wait/MPI_Test` call.

This presents no problem for `MPI_Isends` since they are identified with message sends in the abstract model and the protocol does not require any messages sent before a checkpoint to be resent on restart. However, since for any send request that straddles a checkpoint the application will call `MPI_Wait/MPI_Test` on its `MPI_Request` object, we need to ensure that RRAMP does the right thing for these calls. For `MPI_Wait` RRAMP must return immediately as if the send request

has completed, since it has. For `MPI_Test` must eventually return with a true result (i.e. the request has completed) but not until it has returned with a false result as many times as it did during the original execution, as discussed in detail in Section 2.5.6.2 below.

The case of `MPI_Irecv` is different because it is the `MPI_Wait/MPI_Test` call of a receive request that corresponds to the actual receive, rather than the call to `MPI_Irecv`. Consider a receive request that straddles the checkpoint. If this receive was for an intra-interval message, it needs to be reissued to the MPI implementation via another call to `MPI_Irecv`. This way, when during the restart execution this message is resent and the application calls `MPI_Wait/MPI_Test` to re-receive, it will return correctly when the message actually arrives. Alternately, if this receive request corresponds to a late message, it does not need to be reissued since RRAMP will perform the necessary receive functionality by pulling the message data from the log and writing it to the receive buffer. As discussed in Section 2.5.6.2 below, `MPI_Test` must return with a false result as many times as it did during the original execution.

### 2.5.6.2 Non-determinism Issues

The non-deterministic nature of `MPI_Test` is an important aspect of non-blocking communication. Given the `MPI_Request` object of some send or receive request, `MPI_Test` responds with true or false depending on whether the given request has completed or not. Depending on the timing details of a given run of the application, a given call to `MPI_Test` on a given `MPI_Request` object may non-deterministically report that the request has or has not completed. From the application's point of view this means that the number of times `MPI_Test` responds with false before it

responds with true for a given `MPI_Request` object is non-deterministic.

During the logging period of the protocol each process must record the outcomes of all of its non-deterministic events and deterministically repeat them on restart. To do this for `MPI_Test` we must record the number of times `MPI_Test` responds with false for a given `MPI_Request` object before either the logging period is finished (and we therefore do not need to ensure determinism anymore) or `MPI_Test` responds with true for this request. On restart we can ensure (since `RROMP` wraps all calls to `MPI_Test` with its own code) that each `MPI_Test` will respond with false as many times as were recorded in the log. Non-determinism also appears in functions `MPI_Testall`, `MPI_Waitsome` and `MPI_Testsome` and needs to be recorded and replayed in a similar fashion.

### 2.5.7 Non-deterministic Message Arrival

In addition to the non-determinism in `MPI_Test` and related calls, MPI allows for messages themselves to arrive in a non-deterministic order. In particular, while `MPI_Recv` and `MPI_Irecv` typically use their source, tag and communicator arguments to uniquely identify a FIFO channel to listen on, MPI also provides special wildcard constants `MPI_SOURCE_ANY` and `MPI_TAG_ANY` that allow `MPI_Recv`/`MPI_Irecv` to match messages arriving on channels from any sender and with any tag. Thus, a call to `MPI_Recv` with `MPI_SOURCE_ANY` as the source field will match messages coming in from any process that have the same tag and communicator as the `MPI_Recv` call. There is no wildcard constant for communicators. Since these wildcards allow a single `MPI_Recv`/`MPI_Irecv` call to match messages coming in on multiple FIFO channels, the arrival order of these messages is non-deterministic because there are no ordering guarantees on the arrival order of messages from

different channels.

To log this non-deterministic arrival order, RRAMP records the actual sender and/or tag of any message received by a `RRAMP_Recv/RRAMP_Irecv` call that (i) is executed during the logging period and (ii) uses a `MPI_SOURCE_ANY` or `MPI_TAG_ANY` wildcard. This can be done by examining the status argument to `MPI_Recv/MPI_Irecv` that is executed by `RRAMP_Recv/RRAMP_Irecv` (`MPI_Status` objects have `.MPI_TAG` `.MPI_SOURCE` fields for this purpose). On restart, corresponding calls to `RRAMP_Recv/RRAMP_Irecv` use the real source processor and tag of the original message instead of the `MPI_SOURCE_ANY/MPI_TAG_ANY`. This means that instead of allowing MPI to non-deterministically pick the FIFO channel on which to receive the next message, MPI will be forced to use exactly the same channel it used during the original execution, ensuring deterministic message reception order.

## 2.5.8 Collective Communication

The primary difference between collective and point-to-point communication is that while a given point-to-point communication involves just two processes, a single collective communication can involve any number of processes. This significantly complicates the parallel rollback restart problem. Both the RRAMP checkpointing protocol and other previously explored protocols (Section 2.3) have focused on point-to-point messages. In the past this has not been a problem since prior work on real systems (Section 2.5.2) has generally focused on solutions at the lower levels of the solution space in Figure 2.22, which feature message passing APIs that do not provide anything more complex than point-to-point communication. However, solutions at the MPI specification level or above work on top

of MPI, which is a rich API with collective operations built into it as primitives. This is even true for MPI implementation-level techniques such as in OpenMPI, where even the implementation's internal message passing API contains collective operations as primitives [116].

### 2.5.8.1 Difficulties

The problem posed by collective communication can be seen in Figure 2.28, which shows examples of `MPI_Bcast` interacting with the RROMP checkpointing protocol by crossing the restart line and the log-end line (the line connecting the points on all processes when they each stopped logging). Figure 2.28(a) is an example of an `MPI_Bcast` call where the root process has not yet performed its checkpoint at the time of the call while one of the non-root processes has already checkpointed. On restart that non-root process will call `MPI_Bcast` and it will need to get the correct data. However, on restart the other processes have already completed this `MPI_Bcast`, meaning that they will not call it again. Another difficulty is shown Figure 2.28(b) where a `MPI_Bcast` call crosses the restart line in a different way: the root process has already taken its checkpoint while some non-root process has not. Thus, on restart the root will call `MPI_Bcast` again but the non-root process will not, resulting in an inconsistency in communication. Finally, in Figure 2.28(c) `MPI_Bcast` crosses the log-end line. Since the root process has already finished logging while one non-root process has not, some events in the non-deterministic log may depend on unlogged non-deterministic events that may happen differently on restart. This pollutes the log, resulting in an incorrect restart.

While this example has focused on `MPI_Bcast`'s interactions with the checkpointing protocol from Section 2.5.4, similar issues exist for other collective oper-

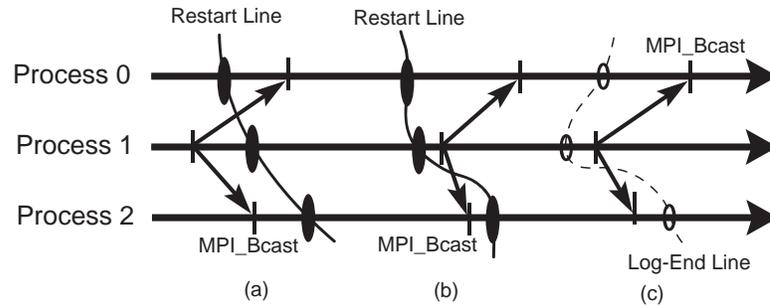


Figure 2.28: MPI\_Bcast interacting with checkpointing protocol

ations and other protocols. The main problems are (i) collective operations have much more complex semantics than their point-to-point counterparts and (ii) a single collective call can involve any number of processes while a point-to-point operation involves only two.

In light of these difficulties one solution to this problem would be to simply ignore the collective operations and reimplement them in terms of point-to-point operations. This would allow us to use any existing rollback restart protocol as if collective operations did not exist. The major drawback of this is that collective operations are generally highly optimized for the details of the native architecture [202] [204] [33], meaning that reimplementing them at a high level with little to no knowledge of the details of the architecture will result in a significant loss in performance.

Since breaking collectives into point-to-points is not desirable, the only efficient solution is to develop protocols that can natively provide rollback restart for collective communication. While this may appear to be difficult or at least difficult to do efficiently, this section describes a way to extend the protocol above to support collective communication. The key to this extension is a detailed description of the semantics of collective calls. Collective operations can be described in terms of (i) how data flows between pairs of processes and/or (ii) real-time synchroniza-

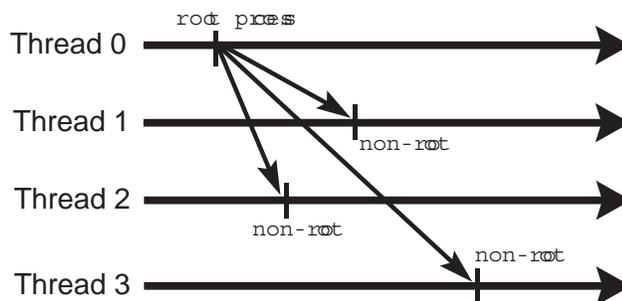


Figure 2.29: Data flows of single-sender collectives

tion constraints between the times when different processes execute the collective operation. In particular, in MPI `MPI_Barrier` is specified via synchronization constraints while all other collective operations are specified via flows of data.

### 2.5.8.2 Semantics of Collectives

Consider the single-sender style of collective operations, `MPI_Bcast` in particular. An `MPI_Bcast` operation on  $n$  processes is defined as  $n$  different point-to-point flows of data. Each flow of data originates at the `MPI_Bcast`'s root process and terminates at the recipient of the broadcast information, as shown in Figure 2.29. Thus, although `MPI_Bcast` is probably not implemented as  $n$  separate point-to-point messages, that is exactly how it is specified. Other single-sender collectives are specified in a similar fashion.

Single-receiver collectives are specified in the reverse fashion, shown in Figure 2.30. Consider the `MPI_Reduce` collective. Every non-root process identifies some information. This information is then sent to the root process, which receives the reduction of the input information via some operation. Thus, for an  $n$  process `MPI_Reduce` operation, there are  $n$  flows of data, one from each non-root process to the root process. The same pattern holds for all the other single-receiver collectives.

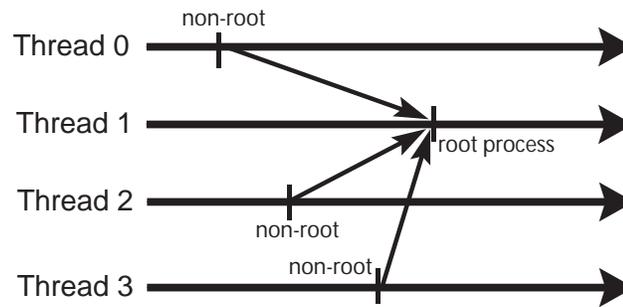


Figure 2.30: Data flows of single-receiver collectives

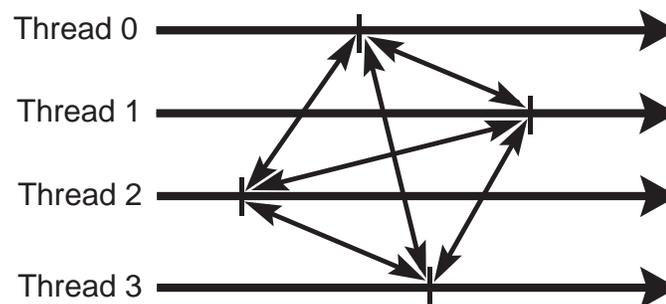


Figure 2.31: Data flows of all-to-all collectives

All-to-all collectives can be described via a pair of data flows between each pair of participants. In the `MPI_AllReduce` collective, each process contributes some data. A reduction of this data is computed and the results are sent to all processes. Thus, an `MPI_Alltoall` operation with  $n$  participants has a total of  $2 * n * (n - 1)$  flows of data, with two flows between every pair of processes, one in each direction. The same is true of all other all-to-all collectives.

`MPI_Barrier` is the only collective operation specified using synchronization rather than data flow semantics. Specifically, when `MPI_Barrier` is called using a particular communicator, no process that is a member of the communicator may exit its `MPI_Barrier` call until every process in the communicator has called it. The idea is to synchronize the execution of code on different processors, even if that code may be using APIs other than MPI. While in practice all-to-all collectives

have the same synchronization properties as barriers, they are not specified to have these properties and as such it is legal to violate them, for example, during restart. Since barriers have synchronization properties by definition, it is never legal to violate them.

### 2.5.8.3 Integrating Collectives into RRAMP

This section discusses the extension of the protocol from Section 2.5.4 to collective operations. This is done by analyzing collective operations in terms of their constituent data flows and applying the same protocol to these data flows as was originally applied just to point-to-point messages. In other words, the data flows used in collective operations are tracked as if they were messages, are counted in the sent messages counters and as part of the counts carried by *mySendCount* messages. Because the goal of this extension is use native collective operations rather than break them up into point-to-point operations, detection of whether a given data flow is early, late or intra-log is performed differently for different types of collective operations (single-sender, single-receiver, etc.), as described below. Ultimately, the guarantees provided by this extension are similar to the guarantees for point-to-point communication of Claim 1:

#### Claim 2

1. *No process can stop logging until all processes are beyond the restart line.*
2. *In a collective communication call, data cannot flow from a process  $p$  that is beyond the log-end line to a process  $q$  that is behind the log-end line.*
3. *In a collective communication call, data cannot flow from a process  $p$  that is behind the restart line to a process  $q$  that is beyond the log-end line.*

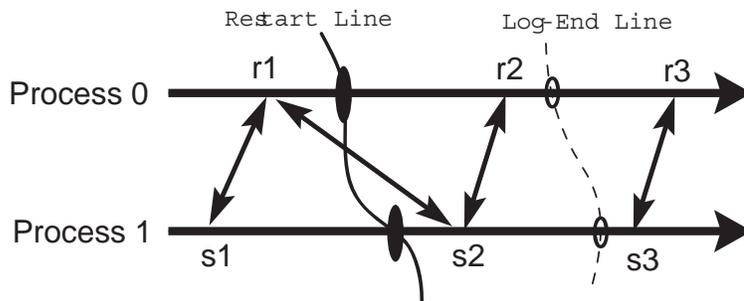


Figure 2.32: Possible types of data flows that may happen in all-to-all collective communications

Therefore, we see that if the arrows in Figure 2.25 are interpreted as directions of data flow, this Figure shows the possible data flows for single-sender and single-receiver collective communication calls. For example, if process  $q$  executes an `MPI_Bcast` at point  $s1$  (before taking its checkpoint), process  $p$  must receive this value before it crosses the log-end line. Data flow in all-to-all communication is symmetric, so the possible data flows are simpler, and are shown in Figure 2.32. For example, a process  $p$  that has crossed the log-end line cannot be involved in an all-to-all collective communication call with a process  $q$  that is beyond the restart line but is still logging.

#### 2.5.8.4 All-to-all collective operations

The protocol for all-to-all collective communication calls is explained using `MPI_Allreduce` as an example. When the RROMP layer intercepts an invocation of `MPI_Allreduce`, it executes the code shown in function `RROMP_Allreduce` in Figure 2.33. This code should be understood with reference to Figure 2.32.

Suppose that the `MPI_Allreduce` straddles the restart line (that is, there are at least two processes between which data flow is of the type  $s2-r1$  in Figure 2.32), corresponding to a late data flow and an early flow. As with late messages, the issue

with late data flows is that the origins of these flows (calls `MPI_Allreduce` that are behind the restart line) will not be re-executed on restart and thus, in order for the destinations of these flows (calls to `MPI_Allreduce` that are beyond the restart line) to get their data, the flow must be logged at checkpoint time and replayed on restart. Early data flows mean that their origins (calls to `MPI_Allreduce` that are beyond the restart line) must be re-executed deterministically on restart, which must happen in this case because all `MPI_Allreduce` calls beyond the restart line are still inside the logging period. To summarize, if an all-to-all call straddles the restart line, the results of any calls beyond the restart line must be recorded at checkpoint time and replayed on restart. Furthermore, the outcomes of all non-deterministic events that precede any straddling `MPI_Allreduce` must be recorded and replayed deterministically on restart.

Now suppose that the `MPI_Allreduce` does not straddle the restart line. If all processes are behind the restart line (the data flows are of the form `s1-r1` in Figure 2.32), no process re-executes the call after restart, so there is nothing to be done. If all processes are beyond the restart line but behind the log-end line (in Figure 2.32, data flows are of the form `s2-r2`), all the data flows are intra-interval and so again there is nothing for us to do since all processes will perform this all-to-all call again on restart. Otherwise, suppose that least one of the processes has stopped logging. If so, in order to ensure that the invariants in Claim 2 hold, this information is propagated to all other processes in the call. When a given process learns that some participant in the `MPI_Allreduce` has stopped logging, it will also stop logging, meaning that the resulting data flows end up being of the form `s3-r3` in Figure 2.32. As such, instead of straddling the log-end line, the `MPI_Allreduce` will happen fully beyond it, enforcing the invariant that no data

```

RROMP_Allreduce(send_data, recv_data, op, comm)
{
    MPI_Allreduce(color, crosses_restart_line, MPI_LXOR, comm);
    MPI_Allreduce(!amLogging, some_not_logging, MPI_LOR, comm);
    MPI_Allreduce(send_data, recv_data, op, comm);
    switch{
        case crosses_restart_line && amLogging:
            log.save(recv_data, comm);
        case !crosses_restart_line && amLogging && some_not_logging:
            amLogging = false;
    }
}

```

Figure 2.33: Protocol for an all-to-all communication

flow crosses the log-end line backwards.

Putting all this together, we see that each process must send its interval color bit and its *amLogging* bit to other processes. By comparing the color bits of other processes participating in the all-to-all, each process can determine whether the call straddled the restart line by checking whether the bits are all equal (did not straddle) or unequal (straddled). Similarly, a process knows that some other process has finished logging if the *amLogging* bit of any other process is false. This logic is implemented by two calls to `MPI_Allreduce` in the pseudo-code in Figure 2.33. These two calls can be trivially combined into a single call; alternatively, the two bits can be piggybacked on the application data payload. The relative overheads of these alternatives are explored experimentally in Section 2.5.10.3.

### 2.5.8.5 Single-receiver collective operations

`MPI_Reduce` is used to illustrate how the protocol handles single receiver collective communication calls. In Figure 2.25, process  $p$  is assumed to be the root for the collective communication call, and the arrows from process  $q$  to process  $p$  show the data flows that can occur. When the root process of the collective communication invokes the call, it is either logging (point r2 in Figure 2.25) or it is not logging (points r1 or r3 in Figure 2.25).

Suppose that the root process is behind the restart line (point r1). If none of the other processes are beyond the restart line, all information flow is of the form s1-r1. In this case there is nothing to be done because no process executes the collective call on restart. Otherwise, suppose that the collective call straddles the restart line, and some of the data flows are of the form s2-r1. Since these are early data flows we know that their send operations (calls to `MPI_Reduce` by non-root processes) must be suppressed on restart. The root process can identify such processes if each process sends the root its interval color and the root sees that while its own *amLogging* flag is false (indicating that it has not yet taken its checkpoint) the interval color of some other process is different from its own. The root records the IDs of any such processes in its *suppress\_log*. During restart, these processes are informed that they must suppress these collective communication calls. The function of the *suppress\_log* is very similar to that of the *earlyIDs* list.

Suppose that the root process is beyond the restart line and is logging (point r2). If the collective communication does not cross the restart line, all processes execute the call during restart and there is nothing to be done. If the collective communication does cross the restart line (there is information flow of the form s1-r2) we will have a late data flow, meaning that some of the sender processes

will not invoke the collective communication call on restart. As such, during checkpointing the root needs to record the data arriving on these late data flows in the log so that it can be replayed on restart. The remaining data flows are intra-epoch, corresponding to `MPI_Reduce` calls will naturally be re-executed on restart. Note that since these flows originate inside the log, they will be executed exactly the same way on restart, producing the same data at the root node. Because we can be sure that the flow data will not change, in this case the root simply saves the data of all incoming data flows (both the late and the intra-interval) at checkpoint time and adds the IDs of all the non-root processes that performed the collective call beyond the restart line to the *suppress\_log*, causing this collective call to not be re-executed on restart and simply replayed from the log.

The final case is when the collective communication call does not cross the restart line, and at least one of the senders has stopped logging. If so, in order to preserve the invariants from 2, the root process must also stop logging. An `MPI_Reduce` operation is used to inform the root whether any of the senders have stopped logging.

The pseudo-code in Figure 2.34 shows two collective communication calls for sending the interval color and *amLogging* bits to the root. As before, these calls can be combined into one; the information can also be piggybacked on the application payload.

```

RROMP_Reduce(send_data, recv_data, op, root_proc, comm)
{
    if (my_proc_id != root_proc) {
        MPI_Gather(color, ..., root_proc, comm);
        MPI_Reduce(!amLogging, ..., MPI_LOR, root_proc, comm);
        MPI_Reduce(send_data, recv_data, op, root_proc, comm);
    } else {
        /* I am receiving the data ... */
        MPI_Gather(..., colors, my_proc_id, comm);
        MPI_Reduce(..., some_not_logging, MPI_LOR,
                    my_proc_id, comm);
        MPI_Reduce(send_data,recv_data,op,my_proc_id,comm);

        bool crosses_restart_line =
            exists { p — p != my_proc_id && colors[p] != color } ;
        switch {
            /* record early sends for suppression */
            case crosses_restart_line && !amLogging:
                { foreach(p in comm where i != my_proc_id)
                    if (colors[p] != color)
                        suppressList.save(comm,p);
                }
            }
        }
    }
}

```

Figure 2.34: Protocol for single-receiver collective communication



### 2.5.8.6 Single-sender collective operations

We use `MPI_Bcast` to illustrate how the protocol handles single-sender collective communication calls. In Figure 2.25, process  $q$  is assumed to be the root for the collective communication call, and the arrows to process  $p$  show the information flows that can occur. When the root process of the collective communication invokes the call, it is either logging (point  $s2$  in Figure 2.25) or it is not logging (points  $s1$  or  $s3$  in Figure 2.25).

If the root process is behind the restart line when it invokes the call (point  $s1$ ), it does not re-execute the call on restart. If the receiving process  $p$  performs the collective call before the restart line (point  $r1$ ), then it is fine since its data flow is intra-epoch and neither the sender nor the receiver will not perform this collective operation on restart. If the receiving process  $p$  performs the collective call while it is logging (point  $r2$ ), its data flow is late and thus process  $p$  must log the value it receives so it can replay this value on restart. To enable  $p$  to discover if its incoming data flow crosses the restart line, the root process  $q$  must broadcast its interval color to the other processes.

Suppose that the root process is logging when it invokes the call (point  $s2$ ). It is possible that one of the receiving processes has stopped logging (point  $r3$ ). To enable it to restart, it is necessary for the root process to re-execute the broadcast during restart. However, it is possible for one of the receiving processes to be behind the restart line (point  $s1$ ). Such a process would not participate in the collective call during restart. To consume the message that would be sent by the root process during restart, the process logs the call in a *re-exec-log*; on restart, calls in the *re-execlog* are invoked with dummy arguments.

Finally, the root process may be beyond the log-end line when it invokes the

collective call. By broadcasting its *amLogging* bit to the other processes, it informs them that it has stopped logging, and they must stop logging as well. Nothing needs to be logged because both the root process and the receivers re-execute the call during restart.

In the code shown in Figure 2.35, the root process uses two calls to `MPI_Bcast` to broadcast its color and *amLogging* bits. As before, these two calls could be combined; the bits could even be piggy-backed on the application payload.

### 2.5.8.7 Barriers

`MPI_Barrier` has a single argument: a communicator that represents the set of processes that need to be synchronizes via a barrier. No process within the communicator may exit from its `MPI_Barrier` call until all processes within the communicator have called `MPI_Barrier` with the same communicator. Since the semantics of `MPI_Barrier` are based on synchronization rather than the flow of data, it can be used to synchronize operations external to MPI and as such, these synchronization semantics need to be preserved even on restart. This becomes impossible if one process checkpoints before a given `MPI_Barrier` call and another process checkpoints afterwards, as shown in Figure 2.36.

In this example processes 1 and 2 call `MPI_Barrier` with a communicator that includes just these processes while all threads participate in a checkpoint. The resulting restart line crosses the `MPI_Barrier` call, meaning that on restart process 1 will restart in a state where it has already passed the barrier while process 2 will restart in a state where it has not yet passed the barrier. This is an illegal state and cannot be allowed to happen in a legal MPI execution. RRMP therefore needs to force or abort checkpoints to make sure that restart lines never cross barriers.

```

RROMP_Bcast(data, root_proc, comm)
{
    if ( my_proc_id == root_proc ) {
        /* I am sending the data ... */
        MPI_Bcast(color, my_proc_id, comm);
        MPI_Bcast(amLogging, my_proc_id, comm);
        MPI_Bcast(data, root_proc, comm);
    } else {
        /* I am receiving the data ... */

        MPI_Bcast(root_color, root_proc, comm);
        MPI_Bcast(root_is_logging, root_proc, comm);
        MPI_Bcast(data, root_proc, comm);

        bool crosses_restart_line = (color != root_color);

        switch {
            /* log late Bcast */
            case crosses_restart_line && amLogging:
                { log.save(recv_data, comm);
                  break;
                }
        }
    }
}

```

Figure 2.35: Protocol for single-sender collective communication

Figure 2.35 (Continued)

```

/* will have to reexec early Bcast */
case crosses_restart_line && !amLogging:
    { reexec_log.save(comm);
      break;
    }
/* turn off logging */
case !crosses_restart_line && amLogging &&
!root_is_logging:
    { amLogging = false;
      break;
    }
}
}

```

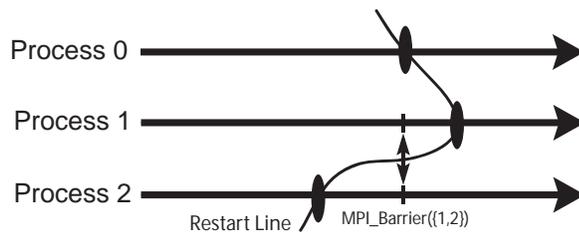


Figure 2.36: Barrier crossing the restart line

```

RROMP_Barrier(comm)
{
    MPI_Allreduce(color, crosses_restart_line, MPI_LXOR, comm);
    if(crosses_restart_line)
        /* force a checkpoint */
        takeCheckpoint();
    MPI_Barrier();
}

```

Figure 2.37: Protocol for barrier communication

Two options are available. If process  $p$  takes a checkpoint while process  $q$  decides to call `MPI_Barrier` that contains  $p$  in its communicator, either (i)  $q$  immediately forces a checkpoint before executing the barrier or (ii) the checkpoint is aborted and the system waits for its next opportunity to take a checkpoint. The disadvantage of the first scheme is that it may force some processes to take a checkpoint at less than optimal locations, potentially increasing checkpoint times. The problem with the second option is worse: partially completed checkpoints are discarded, making the time invested into them useless. Furthermore, there is no guarantee that any checkpoint will ever complete. As such, RROMP uses the checkpoint-forcing approach to dealing with barriers, using the pseudo-code in Figure 2.37.

### 2.5.8.8 Restart

On restart processes exchange their *suppress\_logs* and *re-exec\_logs* just like *earlyIDs* list are exchanged in the base protocol (Section 2.5.4.5). Immediately

afterwards each process checks its *re-exec.log* and re-executes the single-sender collective operations that it needs to participate in as a non-root process. This is done for the sake of collective operations on other processes. These operations are performed with dummy receive buffers since processes performing them do not care about the data they receive.

Each process then resumes its execution in a managed mode that is now extended with the *suppress.log* and additional entries in the log that correspond to collective operations. Whenever the application performs a collective operation in which it participated as a data sender, RRAMP checks whether this call is recorded in its *suppress.log* and if so, removes the entry and does not pass this collective call to the underlying MPI implementation. When the application performs a collective operation in which it is a data receiver, it checks the regular log so see if its data is recorded there. If so, it copies the data from the log directly into its receive buffer, removes the log entry and does not pass the collective operation to the MPI implementation.

Since the application's managed execution is deterministic, we can be sure that each process executes the same sequence of operations on restart as it did during the original execution. Each process exits its managed execution when its log, its *suppress.log* and its *earlyIDs* lists are empty.

## 2.5.9 MPI Opaque State

MPI features a wide variety of opaque objects that the application can use in its interactions with the MPI library. RRAMP's treatment of `MPI_Request` objects is covered in Section 2.5.6. This section covers the mechanism RRAMP uses to deal with the other major object types covered in Section 2.5.1.3: `MPI_Datatype`,

`MPI_Comm`, `MPI_Group` and `MPI_Op`. Remaining types of MPI opaque objects are dealt with in a similar fashion.

### 2.5.9.1 Datatypes, Communicators and Groups

Unlike `MPI_Requests`, objects of type `MPI_Datatype`, `MPI_Comm`, `MPI_Group` are not connected to any given communication and instead carry information that is used in many different communication operations. Although some calls that create these objects may internally involve communication, their independence from individual application communication calls means that it is sufficient for these objects to be recreated by each process before the application is allowed to resume execution.

Datatypes, communicators and groups work in roughly the same way. MPI defines several default datatypes, communicators and groups. It also defines constructor functions that take existing datatypes, communicators and groups as arguments and produce new, modified versions. For example, there is a datatype constructor that given any datatype produces another datatype that corresponds to a contiguous array of data blocks of the original datatype. Another example is a communicator constructor that takes in a communicator as an argument and is called by all processes inside this communicator, each of which provides a numeric color argument. The result of this call on each participating process  $p$  is a communicator that includes process  $p$  as well as all other processes that called the constructor with the same color argument as  $p$ 's. The application defines variables of type `MPI_Datatype`, `MPI_Comm`, `MPI_Group` and uses their respective constructors to write to these variables some data that uniquely identifies their respective objects. The nature of this data is undefined. It may be a full description of the object or merely a unique ID that refers to some data structure inside the MPI

library. The application is allowed to make any number of copies of each object and can use any copy of the same object interchangeably in its interactions with the MPI library.

For checkpointing purposes our goal is to make sure that on restart the application can call MPI functions using its `MPI_Datatype`, `MPI_Comm` and `MPI_Group` objects and get correct behavior from MPI. In order to enable this, the first thing that must be done is to record inside each process's checkpoint the `MPI_Datatype`/`MPI_Comm`/`MPI_Group` objects that existed at that time. On restart MPI's constructor functions can be used to recreate these objects from MPI's point of view. To represent the currently active objects RROMP relies on the hierarchical structure of `MPI_Datatype`, `MPI_Comm`, and `MPI_Group` objects to maintain for each of the three types a "construction tree", containing the creation history of each active object instance.

Figure 2.38 shows a construction tree for datatypes created using the code in Figure 2.39. It maintains representations for `structType`, `intArrayType` and the MPI base types, recording the dependences of one type on others by keeping track of the constructor function required to create the former out of the latter. Since the point of the tree is to make it possible to recreate all active application datatypes, if at some point in time `intArrayType` is freed by the application, the construction tree records its deletion but does not remove it from the tree since `structType`, which is still active, depends on it. However, if `structType` is then also freed, both the `intArrayType` and the `structType` nodes are removed from the construction tree.

At checkpoint time RROMP saves this tree and similar trees for communicators and groups. On restart it restores this tree and uses it to recreate all previously

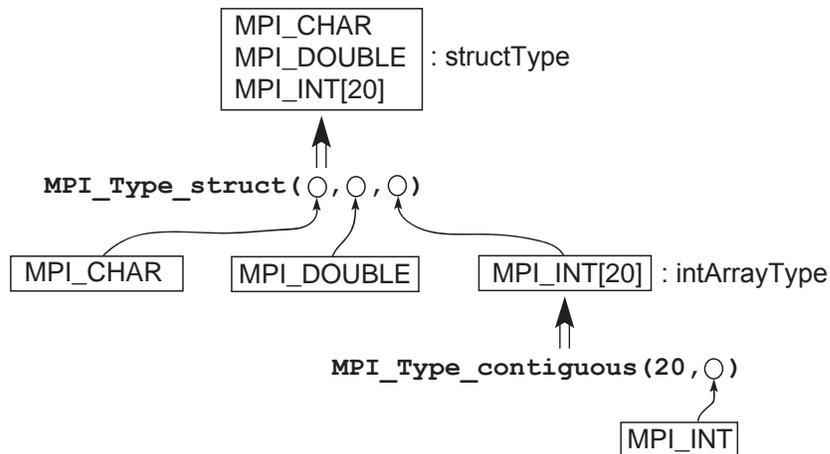


Figure 2.38: Example of a construction tree for datatypes

existing datatypes by starting at the bottom of the tree (always populated by MPI base datatypes) and calling the constructors in the upwards direction until all the datatypes have been recreated. At the end of this procedure, the application's original datatypes will have been recreated inside the restart instance of the MPI library.

While the above procedure recreates the application's original datatypes from MPI's point of view, the application's state must be updated to refer to these new objects rather than the equivalent pre-restart objects. Otherwise, when the application tries to use them in its MPI calls, it will get erroneous results because the original pre-restart objects no longer exist.

The problem is that on restart the application has handles to old instances of its objects while on restart RRAMP has recreated new instances of identical objects. Since on restart it is not possible to update all the copies that the application may have made of its objects with their new values, the solution must be to add a layer of indirection. Instead of giving the application the real `MPI_Datatype`/`MPI_Comm`/`MPI_Group` objects, RRAMP gives it handles to RRAMP objects, each of which

```
MPI.Contiguous(20, MPI.INT, &intArrayType);  
blocklengths[0] = 1;  
blocklengths[1] = 1;  
blocklengths[2] = 1;  
displacements[0] = 0;  
displacements[1] = sizeof(char);  
displacements[2] = sizeof(char)+sizeof(double);  
types[0] = MPI.CHAR;  
types[1] = MPI.DOUBLE;  
types[2] = intArrayType;  
MPI.Struct(3, blocklengths, displacements, types, &structType);
```

Figure 2.39: Datatype creation

contains the original `MPI_Datatype`/`MPI_Comm`/`MPI_Group` objects from the MPI library. The application then uses these `RROMP_Datatype`/`RROMP_Comm`/`RROMP_Group` objects when calling RROMP's wrappers of MPI functions. Each wrapper function verifies whether the MPI object inside the given RROMP wrapper object is fresh before actually using the MPI object to make calls to real MPI library routines.

The freshness check is simple. RROMP maintains a *restartCounter* that is initialized to 0 and is incremented every time the application restarts. In addition to an MPI object, each RROMP wrapper object maintains the value of *restartCounter* from the last time its MPI object was updated with a fresh value. Thus, when an RROMP wrapper function such as `RROMP_Send` is called with some `RROMP_Datatype` and `RROMP_Comm` objects, it first verifies that their values of *restartCounter* are equal to the current *restartCounter* and if not, updates their MPI objects to their current value. It may then use these MPI objects to make actual MPI library calls.

The same indirection mechanism is used for `MPI_Request`, `MPI_Op` and other MPI objects.

### 2.5.9.2 Reduction Operations

Reduction operations are associative and possibly commutative functions used by the `MPI_Reduce`, `MPI_Allreduce`, etc. collective communication functions to compute a reduction of numbers stored in processes. While MPI provides a number of default operations such as `MPI_SUM` and `MPI_MAX`, it is frequently useful for programmers to define their own reduction functions. This is enabled by the function `MPI_Op_create`, which takes a pointer to some user-defined function and produces an `MPI_Op` object that can then be used as a regular reduction operation

in calls to `MPI_Reduce`, `MPI_Allreduce` and `MPI_Scan`. The user-defined function is of type `MPI_User_function(void *invec, void* inoutvec, int* len, MPI_Datatype *datatype)`, meaning that it takes in two buffers of input data `invec` and `inoutvec` of type `datatype`, with `len` data entries in each buffer and writes the result of applying its reduction operation to `inoutvec`.

Recreating `MPI_Op` objects on restart is simple. Each `RROMP_Op` that wraps a given `MPI_Op` object must keep a pointer to the user-defined reduction function. On restart, when the application tries to use this reduction operation, `MPI_Op_create` can be called to recreate the corresponding `MPI_Op` object.

While this mechanism successfully recreates `MPI_Op` objects on restart, the object wrapping mechanism described in the section presents a problem for user-defined reduction functions. The problem is that one of the arguments to the function is of type `MPI_Datatype`. Since the MPI library calls this user function directly during collective reduction operations, it will pass to it a native `MPI_Datatype` object rather than its `RROMP_Datatype` wrapper. If the application then tries to use this `MPI_Datatype` object in any call to datatype-related MPI function, it will cause an error since the MPI function's `RROMP` wrapper will expect an object of type `RROMP_Datatype` rather than `MPI_Datatype` and will therefore fail when it tries to extract an `MPI_Datatype` from what it mistakenly believes to be a `RROMP_Datatype`.

The solution to this problem is to place a layer of indirection between MPI and the user-defined reduction functions. `RROMP_Op_create` takes as an argument a pointer to the original user-defined function. It then dynamically loads a new code file that contains a single global pointer and a shadow reduction function. It then saves in this pointer a reference to the user-defined reduction function and passes

the shadow reduction function to `MPI_Op_create` as if it were the real user-defined function. When the application performs a collective reduction operation MPI will call the shadow reduction function, which maps its `MPI_Datatype` argument to a `RROMP_Datatype` object and passes this object to the actual user-defined reduction function, a pointer to which is held by that code file's global pointer. Thus, this approach wraps each user-defined reduction function with a `RROMP` shadow function that performs the necessary type conversions, generating a new shadow function for each new user-defined reduction function.

### 2.5.10 Experimental Evaluation

A checkpointing system can affect application performance in three ways.

1. It can make the application run more slowly even when no checkpoints are taken.
2. Every checkpoint taken takes up time and resources that could have been used by the original computation, thus slowing it down.
3. When the application needs to be rolled back, it will lose computation time (the amount of time lost depends on checkpointing frequency) and will spend time restarting the application. This restart time is another important parameter.

Depending on desired checkpointing frequency (which can depend on the system failure rate), frequency of restarts and the details of the system and the application, the relative importance of the above numbers changes, requiring us to evaluate these numbers independently. This Section provides an experimental evaluation of the `RROMP` system that measures all of these effects.

### 2.5.10.1 Small-scale Applications Experiment

**2.5.10.1.1 Experimental setup** This experiment was performed on the CMI cluster at the Cornell Velocity supercomputer. This cluster is composed of 64 2-way PentiumIII 1Ghz nodes, featuring 2GB of RAM and connected via a Gigaset switch. The nodes have 40MB/sec bandwidth to local disk. Experiments were performed on 16 nodes, with 1 processor used per node to avoid OS interference. The operating system on the machines was Windows 2000 and we used MPI/Pro 1.6.4 as our MPI implementation. The applications were compiled using the Microsoft C/C++ Optimizing Compiler version 12, using the "Optimized for Speed" optimization setting. RROMP was used for checkpointing the state of MPI while a compiler-based application-level checkpointer [53] was used to checkpoint the state of the application itself.

The performance of this checkpointer was evaluated on three codes:

- A dense Conjugate Gradient code from Yingfeng Su of the University of San Francisco. This code implements a parallel conjugate gradient algorithm with block row distribution. The main loop performs a parallel matrix vector multiply and a parallel dot product, with communication coming from an allReduce and an allGather, which are implemented in terms of point-to-point messages along a butterfly tree. The dense CG code was executed for 500 iterations.
- A Laplace Solver, by Raghu Reddy from the Pittsburgh Supercomputing Center. This program uses an  $n \times n$  grid of numbers that is distributed by block rows. During each iteration every grid cell is updated to be the average of the numbers contained by the neighboring cells (up, down, left, right)

in the previous iteration. The communication comes from each processor exchanging border rows with the processor "above" it and the processor "below" it. The Laplace code was executed for 40000 iterations.

- Neurosys, a neuron simulator by Peter Pacheco of the University of San Francisco (available publically at <http://nexus.cs.usfca.edu/neurosys/>), uses a graph of neurons which excite and inhibit each other via their connections. The current state of each neuron is computed by solving a function of the states of the neurons that are connected to it. The evolution of the neuron network through time is computed via the Runge-Kutta method for differential equations. The program is parallelized by assigning each processor a block of neurons to work with. Communication consists of 5 `MPI_Allgathers` and 1 `MPI_Gather` in each loop iteration. Neurosys was executed for 3000 iterations.

In the following experiments RRAMP was paired with the Cornell Checkpoint Compiler ( $C^3$ ) system [139] [54] [55] an application-level checkpointer that transforms the application source code to make the application self-checkpointing and self-restarting on any platform on which it may run. (a version of  $C^3$  for shared memory is discussed in Chapter 3) In addition to its impact on checkpoint and restart time,  $C^3$  has a checkpoint-free overhead because it adds state logging code and transforms the application code in ways that may reduce the effectiveness of the native compiler's optimizations. The checkpointing overheads reported in this section include the overheads of both  $C^3$  and RRAMP.

All the checkpoints in the experiments are written to the local disk, with a checkpoint interval of 30 seconds.

**2.5.10.1.2 RRAMP Overheads** The performance of RRAMP protocol was measured by recording the runtimes of each of four versions of the above codes.

1. The unmodified program
2. Version #1 + code to piggyback data on messages
3. Version #2 + protocol's logs and saving the MPI library state
4. Version #3 + saving the application state

Experimental results are shown in (Table 2.1). For each code it shows several input sizes. For each input size it shows the size of the checkpoint generated while checkpointing an execution of this configuration. It also shows the running times of all four program versions. For versions #2, #3 and #4 the table shows the overhead of these versions relative to version #1.

- The overhead of using RRAMP without taking any checkpoints is low for both dense CG and Laplace. It is quite high for Neurosys for a small number of neurons as there is very little computation and a large amount of communication. However, the RRAMP checkpoint-free drops to <3% as the input size is increased to a more realistic number. The reason for this is that Neurosys uses 5 `MPI_Allgather`s in every iteration and in our implementation, each such data `MPI_Allgather` is preceded by a command `MPI_Allgather` that sends around the relevant control information. This accounts for the jump in runtime which is as high as 160% for 16x16. However, as the input sizes increase, the message sizes and computation time also increase but the number of messages does not. Thus, the additional work masks the overhead associated with passing around control data, leading this overhead to drop to just 2.7% for a 128x128 neuron grid.

Table 2.1: RROMP, Checkpointing overhead; running times and relative % overheads

Code	Input Size	Checkpoint Size	Original Application	Using RROMP, No Checkpoints	
Dense Conjugate Gradient	4096x4096	8.2MB	117s	119s	1.71%
	8192x8192	33MB	457s	461s	0.88%
	16834x16834	131MB	1815s	1826s	0.61%
Laplace Solver	512x512	138KB	185s	185s	0.0%
	1024x1024	532KB	538s	537s	-0.19%
	2048x2048	2.1MB	2866s	2866s	0.0%
Neurosyst	16x16	18KB	67s	175s	161%
	32x32	75KB	123s	227s	85%
	64x64	308KB	406s	546s	34.5%
	128x128	1.24MB	1948s	2002s	2.77%
Code	Input Size	Checkpoint, No App State		Full Checkpoints	
Dense Conjugate Gradient	4096x4096	134s	14.5%	126s	7.7%
	8192x8192	466s	1.97%	517s	13.1%
	16834x16834	1896s	4.46%	2588s	42.6%
Laplace Solver	512x512	190s	2.7%	189s	2.16%
	1024x1024	538s	0.0%	540s	0.37%
	2048x2048	2907s	1.22%	2897s	1.08%
Neurosyst	16x16	188s	181%	191s	185%
	32x32	246s	100%	247s	100.81%
	64x64	586s	44%	585s	44%
	128x128	2172s	11.5%	2183s	12.1%

- Checkpointing MPI state adds little overhead to dense CG and Laplace. This overhead is higher for Neurosys but is much smaller than the base overhead of RRAMP, even when checkpointing every 30 seconds.
- The cost of recording the application checkpoint varies directly with the size of the checkpoint. Dense CG checkpointing every 30 seconds <14% for 4096x4096 or 8192x8192 matrices. This increases dramatically to 43% when we move up to a 16384x16384 matrix. However, since the overhead is only 4.46% when we do everything but record the application state, it is clear that the reason for the increased overhead is the 131MB of application state. Laplace and Neurosys, both of which have small states show low overheads from recording application state, with the difference between versions #2 and #3 being only a few seconds while checkpointing every 30 seconds.

### 2.5.10.2 Large-scale Applications Experiment

The basic RRAMP protocol described in Section 2.5.4 was extended by Martin Schulz in [189] to enhance its scalability. It differs from the above protocol in two major ways.

First, a processor that has received *sendCount* messages from all of its neighbors knows that all neighbors have taken their checkpoint, meaning that no message that it sends from this point on may be early. However, it is still possible for messages received after this point in time to be late. The Schulz protocol uses this insight to separate the logging period into two segments. The *nonDet/Late* logging segment lasts from the time a process checkpoints and ends when it has received *sendCount* messages from all its neighbors. Since the process may send early messages and receive late messages throughout this time segment, it logs

the outcomes of all non-deterministic decisions and the data of all incoming late messages. The *lateOnly* logging segment lasts from the end of the *nonDet/Late* segment until the process has received all of its late messages. During this segment it only logs the data of incoming late messages. This enhancement speeds up the checkpointing process by reducing the amount of time during which each process logs its non-deterministic decisions.

Second, collective operations are analyzed in terms of their constituent data flows, just like in the protocol above. However, while in the protocol above these data flows are merely used for accounting and to determine what needs to be done to restart collectives, the Schulz protocol takes all collective operations that crossed the restart line and on restart reimplements them in terms of point-to-point messages. Thus, on restart, any messages that correspond to early data flows get suppressed, any messages that correspond to late data flows get read back from the log and any messages that correspond to intra-log data flows get resent and re-received. Note that this break-up into point-to-point messages occurs only on restart and only to collective operations that cross the restart line; it is not done during the application's regular execution. This approach greatly simplifies rollback restart for collective operations and presents future directions for a generic mechanism for using existing point-to-point protocols with collective operations.

This variant of the original RROMP protocol was implemented in the RROMP framework and evaluated at a larger scale than the original protocol to determine its scalability. Given the relatively small differences between the two protocols, this scalability experiment provides a good evaluation of the RROMP protocol presented here as well as the entire RROMP framework. In this experiment RROMP was again paired with the  $C^3$  sequential application-level checkpointer.

**2.5.10.2.1 Experimental Setup** The scalability experiment was performed on the following machines,

**Lemieux** The Lemieux system at the Pittsburgh Supercomputing Center consists of 750 Compaq Alphaserver ES45 nodes. Each node contains four 1-GHz Alpha processors and runs the Tru64 Unix operating system. Each node has 4 GB of memory and 38GB local disk. The nodes are connected with a Quadrics interconnection network.

**Velocity 2** The Velocity 2 cluster at the Cornell Theory Center consists of 128 dual processor 2.4GHz Intel Pentium 4 Xeon nodes. Each processor has a 512KB L2 cache, and runs Windows Advanced Server 2000. Each node has 2 GB of RAM and a 72GB local disk. The nodes are connected with Force10 Gigabit Ethernet.

**CMI** The CMI cluster at the Cornell Theory Center consists of 64 dual processor 1GHz Intel Pentium 3 nodes. Each processor has a 512KB L2 cache, and runs Windows Advanced Server 2000. Each node has 2 GB of RAM and a 18GB local disk. The nodes are connected with a Gigaset switch. This is the cluster used in the small scale experiment.

Unless otherwise notes, the experiments under Windows were performed on Velocity 2. The Windows runs performed on CMI are identified as such.

This experiment focuses on the NAS Parallel Benchmarks (NPB), which are interesting to us because with the exception of the MG benchmark, they do not contain calls to `MPI_Barrier` in the computations. Several of the codes call `MPI_Barrier` immediately before starting and stopping the benchmark timer, but only MG calls `MPI_Barrier` during the computation. All machines are heavily

used by other users, so results could only be obtained for only a subset of the full NAS benchmark set. Also presented are results for the SMG2000 application from the ASCI Purple benchmarks [57] and the HPL benchmark [162].

**Checkpoint placement:**

There are tradeoffs that need to be made when inserting potential checkpoint locations in an application code. On the one hand, if a checkpoint location is specified inside a computation loop, it will be encountered frequently. This tends to reduce the amount of time spent logging since all of the processors are likely to take their checkpoints at roughly the same time. On the other hand, the checkpointing code inserted by the  $C^3$  will add to the execution time of the computation and may inhibit certain compiler optimizations.

The checkpoint locations used in these experiments are documented below. No attempt was made to measure how checkpoint placement impacts performance.

**CG** A checkpoint location is placed at the bottom of the main loop in the routine, `conj_grad`. This loop is the main computational loop of the application.

**LU** A checkpoint location is placed at the bottom of the `istep` loop in the routine, `ssor`. Most of the computations are performed within subroutine calls made within this loop.

**SP** A checkpoint location is placed at the bottom of the `step` loop in the main routine. Almost all of the computations are performed within a subroutine call made within this loop.

**SMG2000** Eight checkpoint locations are placed:

- At the top of the `while i` loop in the routine, `hypre_PCGSolve`.

- At the top of the `for i` loop in the routine, `hypr_ SMGSolve`.
- Five locations are placed in various places throughout the main routine.

These locations represent a mixture of locations both inside and outside main computation loops.

**HPL** A checkpoint location is placed at the top of the innermost driver loop (i.e., `indv`) in `main`. Almost all of the computations are performed within a subroutine call made within this loop.

**2.5.10.2.2 Overhead Without Checkpoints** Tables 2.2 and 2.3 show the overhead of using RRAMP without taking any checkpoints on Lemieux and Velocity 2, respectively. The column labelled “Original” shows the running time in seconds of the original benchmark application. The column labelled “RRAMP+ $C^3$ ” shows the running time in seconds of the application that has been compiled and run using the RRAMP system. For these runs, no checkpoints are taken. The column labelled “Relative” shows the relative overhead of using the  $C^3$  system. This overhead comes from executing the book-keeping code inserted by the  $C^3$  precompiler, and the piggybacking and bookkeeping done by our MPI protocol layer.

The overheads on Lemieux are less than 10% on all codes; for most codes in fact, the overheads are within the noise margins, which was around 2-3%. Moreover, there is no particular correlation of overheads to the number of processors, showing that the protocol scales at least to a thousand processors.

The overheads on Velocity 2 are mostly within the 10% range as well, except for `smg2000`. This code has overheads of approximately 50%. The reason for this is the cost of piggybacking the 3 bits of protocol information on top of all

Table 2.2: Runtimes in seconds on Lemieux without checkpoints

Code (Class)	Procs (Nodes)	Runtime		Relative Overhead
		Original	RROMP+C <sup>3</sup>	
CG (D)	64 (16)	1651	1679	1.7%
	256 (64)	447	466	4.2%
	1024 (256)	207	213	3.0%
LU (D)	64 (16)	1500	1571	4.7%
	256 (64)	408	425	4.3%
	1024 (256)	126	134	6.3%
SP (D)	64 (16)	3011	3130	4.0%
	256 (64)	6423	661	2.9%
	1024 (256)	199	205	3.3%
SMG2000	64 (16)	136	143	5.3%
	256 (64)	145	156	7.6%
	1024 (256)	158	172	8.7%
HPL	64 (16)	280	286	2.2%
	256 (64)	-*	-*	-*
	1024 (256)	379	415	9.6%

\*These results were unavailable at the time of publication

Table 2.3: Runtimes in seconds on Velocity 2 without checkpoints

Code (Class)	Procs (Nodes)	Runtime		Relative Overhead
		Original	RROMP+ $C^3$	
CG (D)	64 (32)	4085	4295	5.1%
	128 (64)	1691	1829	8.2%
	256 (128)	1651	1815	9.9%
LU (D)	64 (32)	3232	3284	1.6%
	128 (64)	1814	1908	5.2%
	256 (128)	1074	1108	3.2%
SP (D)	64 (32)	4223	4307	2.0%
	144 (72)	2102	2152	2.4%
	256 (128)	2564	2680	4.5%
SMG2000	32 (16)	231	340	47.6%
	64 (32)	270	420	55.2%
	128 (64)	330	487	47.5%
HPL <sup>†</sup>	32 (16)	3121	3133	0.38%
	64 (32)	1776	1780	0.22%
	128 (64)	1164	1165	0.11%

<sup>†</sup>These runs were performed on CMI

communication. MPI provides no native piggybacking capabilities and as such, RRAMP uses a number of techniques to piggyback protocol information using just the MPI API. The piggybacking mechanism used in RRAMP for this code during this experiment (called "datatype" piggybacking) happens to have a very high overhead for smg2000 on this platform. Section 2.5.10.3 explains this in more detail.

**2.5.10.2.3 Overhead With Checkpoints** The next set of experiments are designed to measure the additional overhead of taking checkpoints.

Tables 2.4 and 2.5 show the run-times and absolute overheads in seconds of taking checkpoints for the same applications shown in Tables 2.2 and 2.3. The meaning of the configurations is as follows.

**Configuration #1.** The run-times of the  $C^3$  generated code using RRAMP without taking any checkpoints. These run-times are the same as shown in column "RRAMP+ $C^3$ " of Tables 2.2 and 2.3.

**Configuration #2.** The run-times of the generated code when computing one checkpoint during the run but without saving any checkpoint data to disk. This highlights RRAMP's contribution to the cost of checkpointing.

**Configuration #3.** This configuration is the same as #2, except that it includes the cost of saving application state to the local disk on each node.

**Checkpoint cost** This is the cost of initiating and taking a single checkpoint, where the base line is Configuration #1. Therefore, this cost does not include the continuous overhead introduced by the  $C^3$  system, shown in Tables 2.2 and 2.3.

Table 2.4: Runtimes in seconds on Lemieux with checkpoints

Code (Class)	Procs (Nodes)	Runtime Configurations			Checkpoint	
		#1	#2	#3	Size/proc. (Mb's)	Cost (secs.)
CG (D)	64 (16)	1679	1703	1705	652.02	26
	256 (64)	466	479	511	244.50	45
	1024 (256)	213	218	237	123.67	24
LU (D)	64 (16)	1571	1543	1554	190.66	-17
	256 (64)	425	425	424	56.83	-1
	1024 (256)	134	143	148	18.38	14
SP (D)	64 (16)	3130	3038	3264	422.85	134
	256 (64)	661	659	678	133.55	17
	1024 (256)	205	215	212	49.27	7
SMG2000	64 (16)	143	143	145	2.88	2
	256 (64)	156	160	159	3.24	3
	1024 (256)	172	183	183	3.60	11
HPL	64 (16)	286	285	285	0.02	0
	256 (64)	-*	-*	-*	-*	-*
	1024 (256)	415	393	396	0.43	-19

\*These results were unavailable at the time of publication

Table 2.5: Runtimes in seconds on Velocity 2 with checkpoints

Code (Class)	Procs (Nodes)	Runtime			Checkpoint	
		Configurations			Size/proc. (Mb's)	Cost (secs.)
		#1	#2	#3		
CG (D)	64 (32)	4295	4296	4304	455.60	9
	128 (64)	1829	1827	1896	246.84	67
	256 (128)	1815	1804	1860	169.25	45
LU (D)	64 (32)	3284	3271	3315	190.57	31
	128 (64)	1908	1874	1901	104.86	-7
	256 (128)	1108	1121	1146	56.83	38
SP (D)	64 (32)	4307	-*	4423	422.76	116
	144 (72)	2152	-*	2231	217.76	79
	256 (128)	2680	-*	2688	133.64	8
SMG2000	32 (16)	340	333	338	506.41	-2
	64 (32)	420	396	408	510.62	-12
	128 (64)	487	493	541	465.65	54
HPL <sup>†</sup>	32 (16)	3133	3136	3140	0.34	7
	64 (32)	1780	1775	1781	0.34	1
	128 (64)	1165	1163	1177	0.34	12

\*These results were unavailable at the time of publication

†These runs were performed on CMI

The difference between Configurations #2 and #1 is that #2 includes the cost of going through the motions of taking a checkpoint without actually saving anything to disk whereas #1 never begins a checkpoint. As such, it primarily consists of RROMP's contribution to the cost of checkpointing. The extra work of Configurations #3 on top of #2 is that #3 includes the cost of saving the checkpoint data to the local disk on each node. Thus, it is  $C^3$ 's contribution to the cost of checkpointing since  $C^3$ 's primary contribution to the cost of taking a checkpoint is the amount of checkpoint data that needs to be saved.

The numbers in Tables 2.4 and 2.5 were measured using a single experiment for each data point. Although it would have been better to repeat each experiment several times and then report the average, it was not possible to get enough time on the machines to accomplish this in time. As mentioned before, the noise margin is about 2-3%. We believe that this accounts for the negative numbers in the last columns of these tables.

These results show that the cost of taking a checkpoint is small. To put these results into perspective, if we scale the running times appropriately, then we see that the maximum overhead when checkpointing once an hour is less than 4% and the maximum overhead when checkpointing once a day is less than .2%.

As we mentioned earlier, Configuration #3 measures the cost of writing the application state to each node's local disk. In a production system, writing checkpoint files to local disk does not ensure fault-tolerance, because when a node is inaccessible, its local disk usually is too. However, writing directly to a non-local disk is usually not a good idea because the network contention and communication to off-cluster resources can add significant overhead. A better strategy that is used by some systems is for the application to write checkpoints to a local disk

and then for an external daemon to asynchronously transfer these checkpoints from local disk to an off-cluster disk. Very often a second, possibly lower performance, network is used to avoid contention with the application’s messages. Such a system has been implemented at the Pittsburgh Supercomputing Center [198], and we have started work to integrate  $C^3$  with that system.

**2.5.10.2.4 Restart Cost** For technical reasons, obtaining accurate measurements of restart costs for the parallel applications proved to be exceptionally difficult, and these results are not available. Nevertheless, it was possible to obtain restart costs for single processor runs of the applications.

Restart cost was computed using the following formula:

$$\text{RestartCost} = \text{BefRestarting} - \text{AftChkpt}.$$

*AftChkpt* = time from the end of the application’s last checkpoint until the end of of the application’s main computation region.

*BefRestarting* = time from the beginning of the restart execution until the end of the application’s main computation region.

The results in seconds are reported in the “Restart Cost, absolute” column of Tables 2.6 and 2.7. To put these results into context, column “Restart Cost, relative” gives these times as a percentage of the “Original” runtime of the unmodified application.

These times are, with one exception, all less than 2% of the execution time of the program, so we consider the restart costs to be negligible.

**2.5.10.2.5 Discussion** The large scale experiment shows that the overhead added by the RROMP system with the  $C^3$  sequential checkpointer as well the cost of taking checkpoints with these systems are fairly small. One reason for

Table 2.6: Restart costs in seconds on Lemieux

Code (Class)	Runtime Original	Restart Cost	
		absolute	relative
CG (A)	13	0	1.8%
LU (A)	244	-5	-1.9%
SP (A)	405	2	0.4%
SMG2000	83	5	5.3%
HPL	231	0	0.1%

Table 2.7: Restart costs in seconds on CMI

Code (Class)	Runtime Original	Restart Cost	
		absolute	relative
CG (A)	34	0	0.5%
LU (A)	900	10	1.1%
SP (A)	1283	-5	-0.4%
SMG2000	172	-1	-0.8%
HPL	831	0	0.1%

this is that the benchmarks considered here do not save much checkpoint data. For benchmarks that save large amounts of data, the cost of writing the data to disk can be significant, especially if the disk is on a network. Compiler analysis to reduce the amount of saved state is one possible solution along with runtime techniques such as incremental checkpointing [166] and compression [170].

### 2.5.10.3 Piggybacking Overhead

The checkpoint-free overhead of RROMP comes in three forms: (i) overhead of wrapping all MPI function and objects with RROMP functions and objects, (ii) overhead of the protocol logic and (iii) overhead of piggybacking information on top of messages. Of the three the last one is the most significant and is explored further in this section.

On the surface it does not appear that piggybacking should have any real overhead in RROMP since RROMP piggybacks only 32 bits onto point-to-point messages (reduced in just 3 bits in [189]) and communicates only 2 bits per process during collective communications. However, recall that RROMP works above the MPI implementation, meaning that it does not have access to any low-level data structures, such as packet headers. As such, even if there is space in packet headers to store this piggybacked information, RROMP cannot use it without violating the abstraction provided by the MPI specification and losing its ability to work with arbitrary MPI implementations. As such, RROMP needs to implement its piggybacking using only the functionality available in the MPI specification.

RROMP uses a variety of mechanisms to piggyback control information on top of application communication. In the case of collective communication, the algorithms for piggybacking RROMP data are detailed in Sections 2.5.8.4, 2.5.8.5

and 2.5.8.6. In the case point-to-point communication RROMP uses three different mechanisms:

- **Separate:** Each application message is immediately followed on the same FIFO channel by a message carrying the message's piggybacked data. Thus, the number of messages sent by the application doubles, although the new messages carry only 32 bits of data.
- **Memcpy:** For each application message RROMP copies the message and its piggybacked data into a separate buffer and sends that buffer instead. The application and piggybacked data are separated out by the receiver.
- **Datatype:** For each application message RROMP creates a new `MPI_Datatype` using the `MPI_Type_struct` constructor. This new type refers to two buffers: the buffer containing the piggyback data and the application buffer. The message is sent using the new datatype, allowing the MPI library choose the best way to send the two buffers in the same message. Datatypes are used at the receiver in a similar way to extract the application data and the piggybacked data.

That piggybacking can have a significant effect on the performance of real applications can be seen by looking at the overhead induced by piggybacking on the `smg2000` benchmark [57] from the ASCI Purple suite. Figure 2.40 shows the overhead of running `smg2000` on processors of the Velocity 1 cluster, CMI and Lemieux, using datatype piggybacking. (Velocity 1 is a cluster at the Cornell Theory Center that is just like CMI, except that each node has 4 500MHz Pentium 3 processors with 4GB RAM per node and 2MB L2 cache per processor). The graphs show that on the two Windows clusters the overhead imposed by datatype

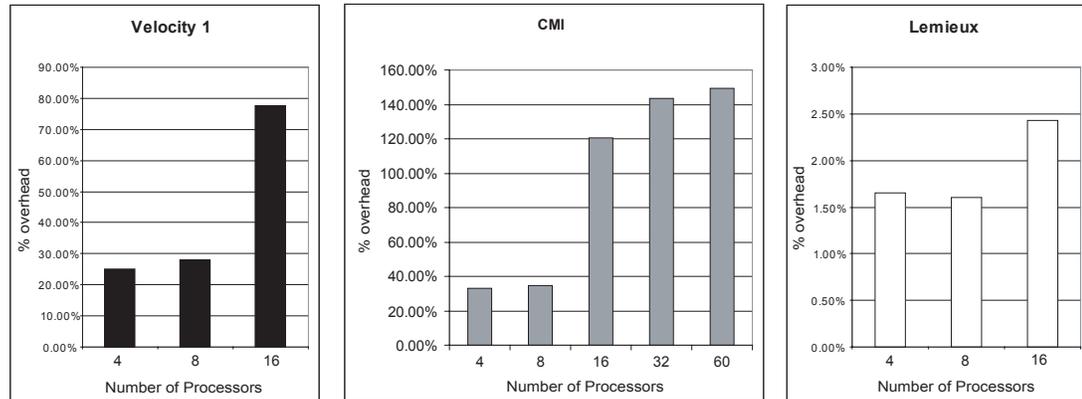


Figure 2.40: Datatype piggybacking overheads for SMG2000

piggybacking on to smg2000 is very high, growing to 140% on CMI as the number of processors reaches 60. Meanwhile the same experiments yield a very low 1%-2.5% overhead on Lemieux. While it is not clear what details of network setup or MPI implementation cause these differences, it is clear that the overheads of piggybacking are both complex and can potentially have a significant effect on application performance.

Furthermore, this experiment helps explain the very high overheads of smg2000 in the large-application experiment above, which used datatype piggybacking. While the experiments in Figure 2.40 were performed on CMI and Velocity 1, rather than Velocity 2, the three systems are quite similar and this experiment underscores the importance of choosing the appropriate type of piggybacking for each application and platform combination.

The rest of this section focuses more closely on the overheads of piggybacking. Section 2.5.10.3.1 discusses its effects on point-to-point communication, Section 2.5.10.3.2 looks at piggybacking on top of collective communication and Section 2.5.10.3.3 focuses on how piggybacking impacts application performance.

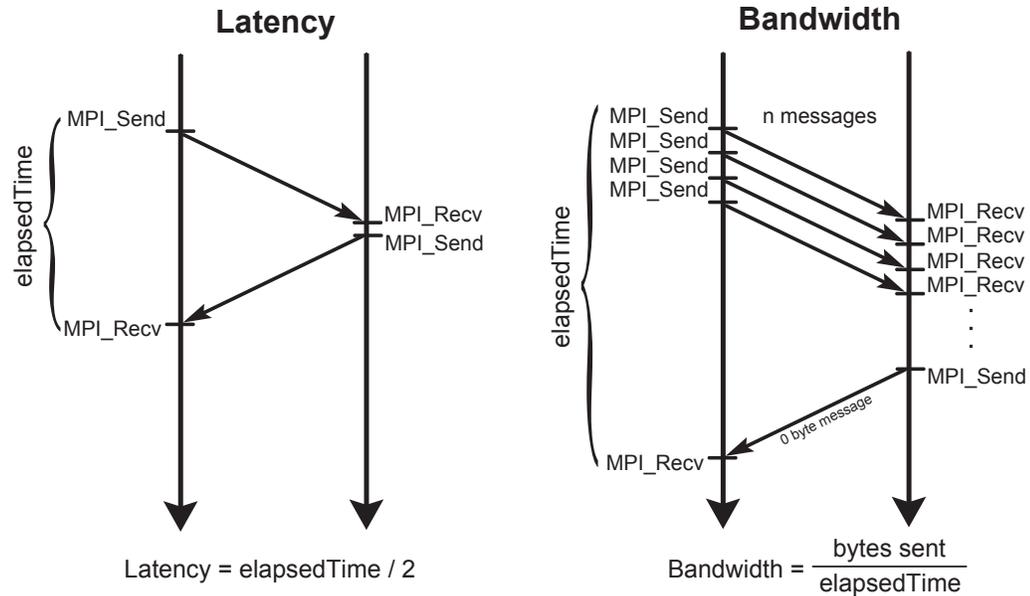


Figure 2.41: Benchmark diagrams

### 2.5.10.3.1 Point-to-point Piggybacking

#### The Benchmarks

Experiments to evaluate the cost of piggybacking on point-to-point messages were performed using the `presta1.2` [12] benchmark from the ASCI Purple benchmark suite [13]. This benchmark contains several low-level messaging tests. The two relevant tests for this evaluation are `com` and `laten`, which measure network bandwidth and latency, respectively.

Figure 2.41 shows the algorithms used to measure the latency and bandwidth of point-to-point communication. Latency is measured by having one processor send a message to another and having the second processor reply with a message of the same size. The first processor measures the amount of time elapsed between its send and its receive and divides it by 2. Bandwidth is measured by having one processor send another a large number of messages. When the second processor has received that many messages, it responds with a 0-byte acknowledgement mes-

sage. The first processor computes the bandwidth by dividing the total amount of data sent by the amount of time between its first send and its receive. While these measurements are not as precise as those used by the LogP model [68] or its variants, they are widely used and allow for meaningful comparisons between different networks.

Because network load changes the latency and bandwidth of real networks, the `com` and `laten` benchmarks try to capture this effect by performing their measurements while varying three parameters. First, given a system with  $m$  processors, where  $2^n \leq m < s^{n+1}$  for some  $n$ , the benchmarks run their respective tests on 2 processors, then on 4, 8, ...,  $2^n$ . Furthermore, the `com` benchmark can be run in both unidirectional and bidirectional mode. Unidirectional mode uses the algorithm from Figure 2.41 while bidirectional mode runs the same algorithm in both directions. Finally, the size of the messages sent can be varied in powers of 2 from 0 bytes to several MB. Although the original version of `laten` does not allow for message sizes to be varied (it always sends 0-byte messages), in these experiments a modified version was used that allows for message size variability.

### **Experimental Results**

The expected overheads of piggybacking 4 bytes of data on various sizes of application messages are shown in Figure 2.42. For small messages the cost of copying a few bytes of message data and control data into a separate buffer is dwarfed by the much higher cost of network transfer. As such, it would be expected that the bandwidth and latency overheads of `memcpy` would be low for small messages. On the other hand, since separate piggybacking sends two separate messages for every application message, this tactic should have a higher bandwidth

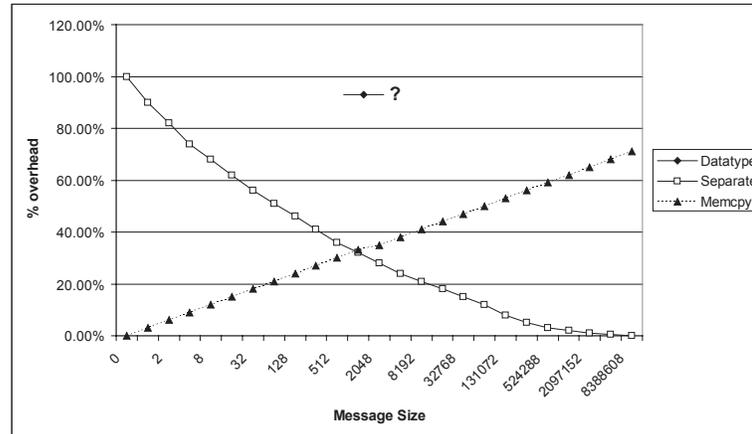


Figure 2.42: Expected piggybacking bandwidth overheads

and latency overhead than memcpy. For large messages this relationship would be expected to switch. Large memcpys are expensive and as such, piggybacking only a few bytes onto an already large message should result in a fairly high overhead since it would involve copying the entire large message. At the same time, given the few bytes being piggybacked, the cost of sending them in a separate message should be dwarfed by the cost of sending the associated large application message. As such, the overhead of separate piggybacking should be low for large messages. These relationships would be expected to hold regardless of network load, although the absolute overheads as well as the exact message size where memcpy and separate cross over should vary depending on the details of the experiment. Since the implementation of datatypes is implementation-defined, it cannot be known a priori how datatype piggybacking will behave.

To determine the actual latency and bandwidth overheads and compare them to the above model, experiments were performed using the `com` and `laten` benchmarks on the Lemieux system. Since Lemieux a cluster of 750 4-way SMP nodes, the intra-node network is different from the inter-node network and must be evaluated separately. Figures 2.43 and 2.44 show the Bandwidth overheads of piggybacking

using datatype, separate and memcpy relative to not using any piggybacking at all (i.e. % reduction in bandwidth). Figure 2.43 shows this for unidirectional bandwidth tests while Figure 2.44 does so for bidirectional tests, in both cases on 2 and 4 processors within the same Lemieux node.

As expected, this experiment shows that in each case there is a point between 4KB and 32KB where memcpy piggybacking performs better than separate before the point and worse afterwards. One surprising feature is that the difference between memcpy and separate is very small for the smaller messages sizes. It may be Lemieux's intra-node network is capable of simultaneously carrying large numbers of small messages, meaning that doubling the number of messages on this network results in little performance degradation. However, this does not explain the fact that there exists a spike after 8 byte messages for 2 processor bidirectional tests but not the 4 processor tests. The ultimate mechanism behind this effect is unclear.

Another unexpected feature is the fact that for intermediate-sized messages (512bytes-8KB for unidirectional and 32bytes-4KB for bidirectional) the bandwidth for piggybacked communication improves significantly relative to non-piggybacked communication. For three of the four configurations it actually causes piggybacking to improve the bandwidth of Lemieux's intra-node network. This effect happens because while bandwidth generally improves with larger message sizes, during this region it improves faster for messages with piggybacking enabled than for those where it is not. Given how quickly non-piggybacked communication overtakes piggybacked communication and the subsequent rise in the overhead of datatype and memcpy piggybacked communication, it appears that non-piggybacked communication suffers from an ill-placed switch between MPI using an eager and rendezvous protocols [99] in its implementation of `MPI_Send`.

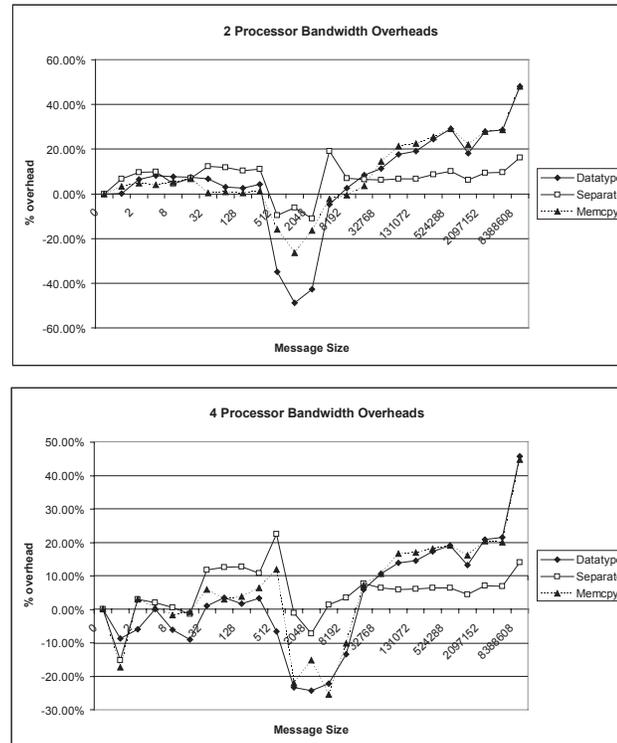


Figure 2.43: Piggybacking bandwidth overheads (intra-node, unidirectional)

Because the messages with piggybacking look different to MPI, it likely shifts their protocol at a different message size.

Despite some deviations, the overhead curve for datatype piggybacking appears to follow the memcpy curve, making it likely that Lemieux’s implementation of MPI internally uses a variant of the memcpy mechanism to implement datatype piggybacking, at least for the intra-node network.

Figure 2.45 shows the results of the latency experiment on 2 and 4 processors within the same Lemieux node. The measured latency overheads due to piggybacking match the expected results precisely, with the memcpy/separate switch-over coming around 10KB messages. Again, the behavior of datatype appears to mimic memcpy but the deviations suggest that the internal MPI implementation

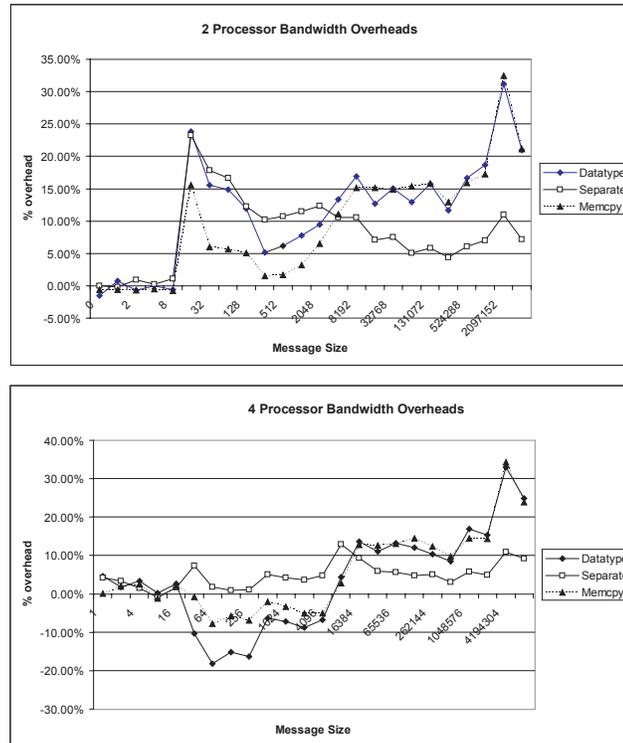


Figure 2.44: Piggybacking bandwidth overheads (intra-node, bidirectional)

of datatype does something somewhat different. Strong possibilities are that the deviation comes from (i) the fact that MPI needs to parse the RRAMP-defined datatype while memcpy does not and (ii) MPI may be able to take advantage of some network card copying features for messages of certain sizes, while memcpy may not.

The above experiments were conducted on Lemieux’s inter-node network. Since each node contains 4 processors, it was possible to execute each test simultaneously on 2, 4 or 8 processors and still ensure that all communication was being carried over the intra-node network. The results from this network are qualitatively similar to those on the intra-node network. In most cases memcpy features low overhead and separate high overhead for small messages. As the message sizes increase the overhead for memcpy rises and overhead for separate falls until the two cross at

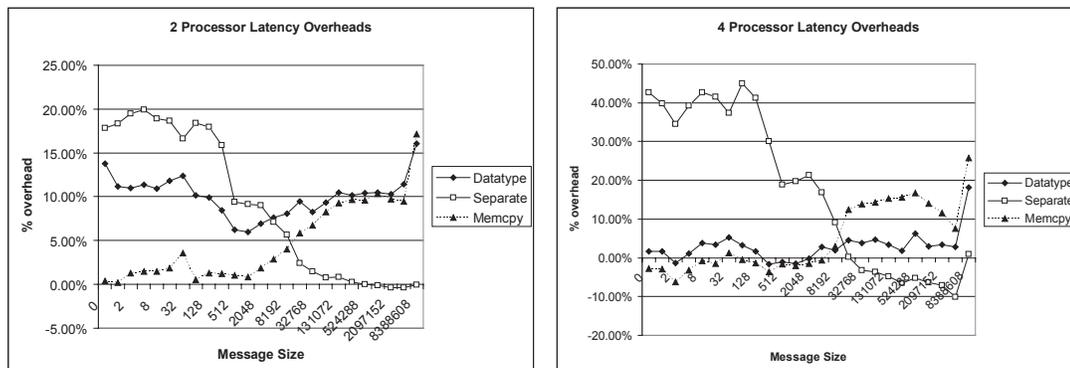


Figure 2.45: Piggybacking latency overheads (intra-node)

some point. Datatype tends to perform somewhere in between the two, sometimes mirroring the overhead curves for memcopy and separate and sometimes not. Thus, it is not clear how datatype is being implemented internally by this implementation of MPI on the inter-node network. There is one interesting effect that is seen in the inter-node network that is not seen on the intra-node network: in some configurations the bandwidth and latency of both memcopy and datatype stay near 0% even for the large message sizes. This effect, shown in Figure 2.46, tends to happen when the load on the network is high (many processors performing the test with large messages), suggesting that the additional overhead of piggybacking is masked by network contention. When a network is congested, processors spend large fractions of their time waiting for the network to be ready to accept its next communication. It appears that this helps to reduce the overhead of piggybacking since the processor is able to perform all the necessary copying operations during these gaps, resulting in almost no additional cost when the processor is finally able to send the message.

**2.5.10.3.2 Collective Piggybacking** The overhead of piggybacking on top of collective operations was measured on the CMI cluster. From each of the sets

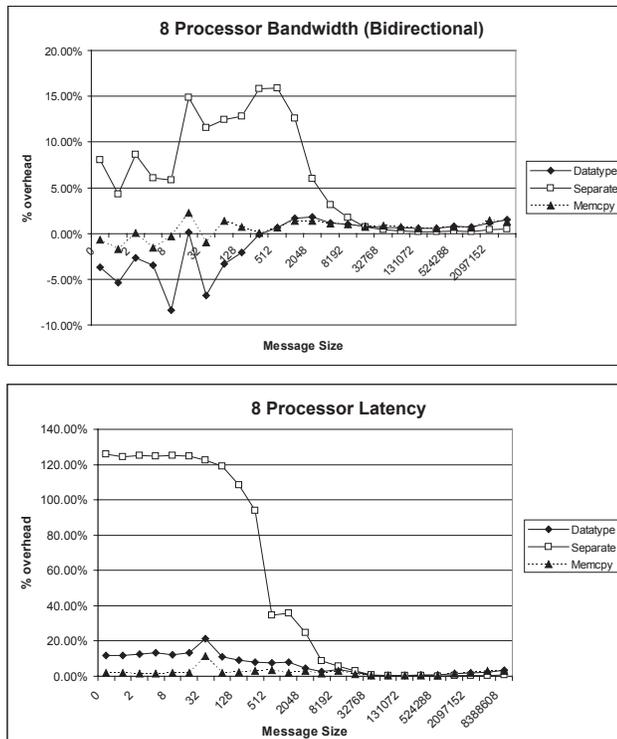


Figure 2.46: Piggybacking latency overheads (intra-node)

of MPI collective calls (single-sender, single-receiver, all-to-all, and barrier) a representative operation was selected (`MPI_Bcast`, `MPI_Gather`, `MPI_Allgather`, and `MPI_Barrier` respectively) and the performance of the native, piggyback-free version was compared to that of the version modified to utilize the basic RROMP protocol. These modifications include sending the necessary protocol data (epoch and logging bits) and performing the protocol logic (as illustrated in the pseudocode) after the data was received. The epoch and logging bits were sent together as a one byte block.

For each of the above calls there are two natural ways to send the protocol data: either via a separate collective operation that precedes the data operation, or by “piggy-backing” the control data onto the message data and sending both with one operation. For comparison purposes, we implemented both methods. The

overhead for the separate operation case includes the time to send both messages. For the combined case, it includes the time to copy the message data and the control data to a contiguous region, the time for the single communication, and the time to separate the message and protocol data on receipt.

The top graph in Figure 2.47 shows the absolute time taken by the native and protocol (both the separate and combined message) versions of `MPI_Bcast` for data message ranging in size from 4 bytes to 4 MB. The bottom graph shows the overhead, in seconds, that the two versions of the protocol add to the communication. Figure 2.48 shows similar information for `MPI_Gather` and Figure 2.49 does so for `MPI_Allgather`. For these calls, message size refers to the size of the data that each of the processes contributes to the collective operation. (Notice that the y-axis has a logarithmic scale.) For small message sizes, the relative overhead (percentage) might be high, but the absolute overhead is very small. For large messages sizes, the absolute overhead might be large, but relative to the cost of the native version, the cost is very small.

Examining the second graph in each set, we observe that the cost of using the separate message protocol is fairly constant, whereas the cost of the combined protocol grows linearly with the size of the message. These behaviors are to be expected: using a separate message imposes a fixed cost, regardless of the size of the data message, while using a combined message requires copying at both the sender(s) and the receiver(s). Therefore, the optimal strategy would be a protocol that switched from a combined message to separate messages as the size of the data message grew. Using such a strategy, the overhead added by this protocol is minimal.

Although a collective communication, `MPI_Barrier` does not actually commu-

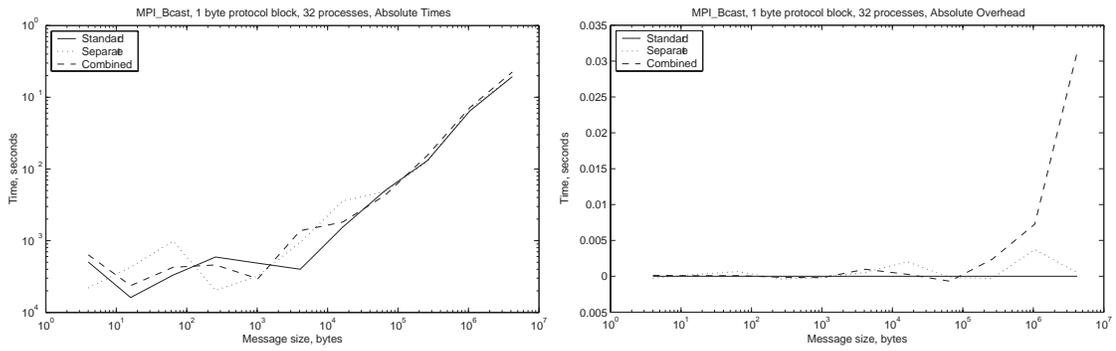


Figure 2.47: MPI\_Bcast

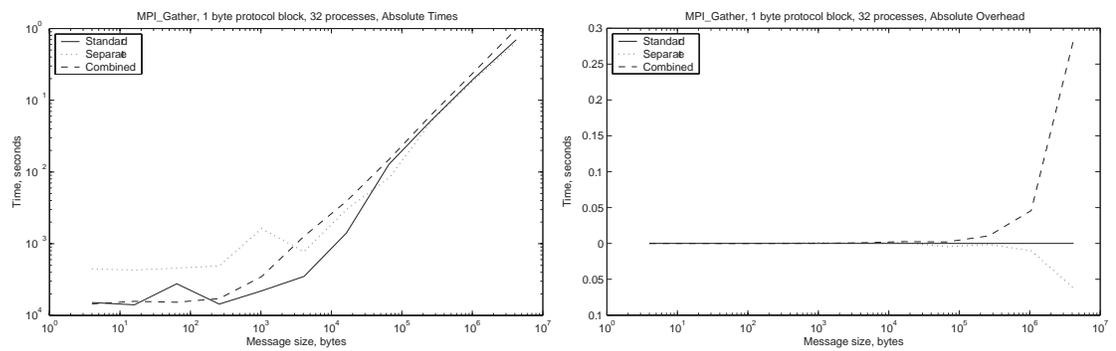


Figure 2.48: MPI\_Gather

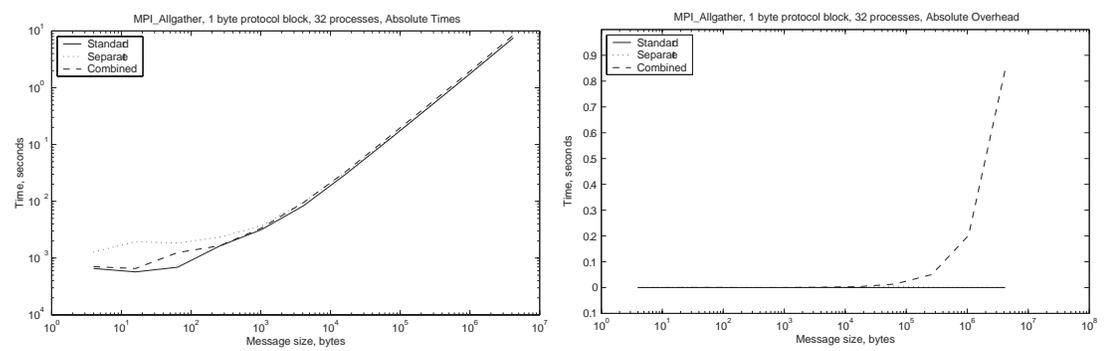


Figure 2.49: MPI\_Allgather

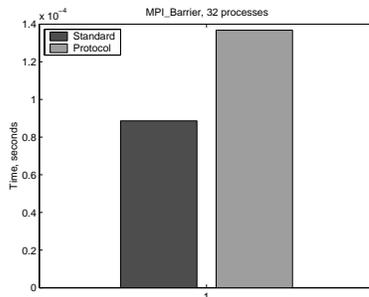


Figure 2.50: MPI\_Barrier

nicate any message data to the application processes. Therefore, we do not have the option of “piggy-backing” the protocol data, and we must use a separate communication to send it. Additionally, experimental results for barrier do not depend on message size.

A chart comparing the performance of the native `MPI_Barrier` versus the protocol version is shown in figure 2.50. The true cost of a barrier is the cost of waiting for all the processes to arrive at it: this cost is much greater than the cost of actual communication that the barrier requires. Our experiments only compare the communication cost of both the native and the protocol version of `MPI_Barrier`, our protocol will not affect synchronization time. The actual difference in the communication time between the two versions is inconsequential.

**2.5.10.3.3 Piggybacking and Application Performance** The effect of piggybacking on real application performance varies significantly with the details of an application. Running a given application on a different input size or a different number of processors can have a significant effect on the effect of piggybacking on its performance. These effects can be seen in Figures 2.51, 2.52 and Figure 2.54. Figures 2.51 and 2.52 show the overheads from using datatype and separate piggybacking with the NAS benchmarks (input classes W and A), HPL, smg2000 and sweep3D (from the ASCI Purple benchmarks) on Velocity 1 with Figures 2.51

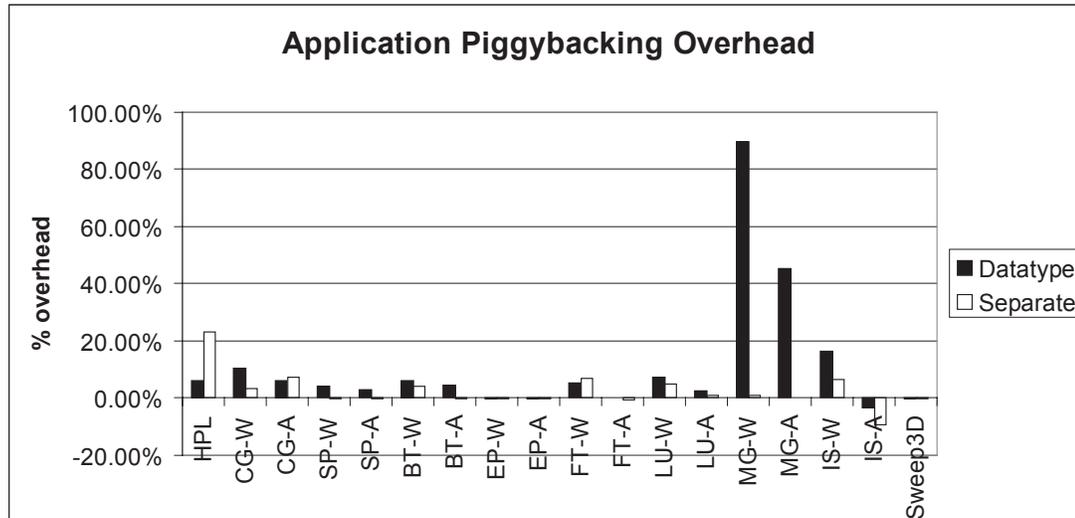


Figure 2.51: Application piggybacking overhead on 4 processors, Velocity 1 showing the results of 4 processor runs and 2.52 showing 16 processor runs. Figures 2.53, 2.54 and 2.55 show the overheads of datatype, separate and memcpy piggybacking on CG (Figure 2.53), LU (Figure 2.54) and HPL(Figure 2.55) running on 4 - 512 Lemieux, using piggybacking for point-to-point messages but not collective communication. CG and LU were run on input class C.

These graphs feature a number of interesting effects:

- No piggybacking mechanism is best for all applications or even for a single application. LU-C on Lemieux (Figure 2.53) is an example of this, showing that while separate piggybacking performs best on 4-64 processors with this code, memcpy is best for 128 and 256 processors and datatype wins for 512 processors. For an example with larger overall overheads consider HPL on Lemieux (Figure 2.55), where memcpy is the general winner, except for the 64 and 512 processor runs.
- The overhead can vary dramatically between two different input sizes for the same application. In particular, consider FT on Velocity 1, run on 16

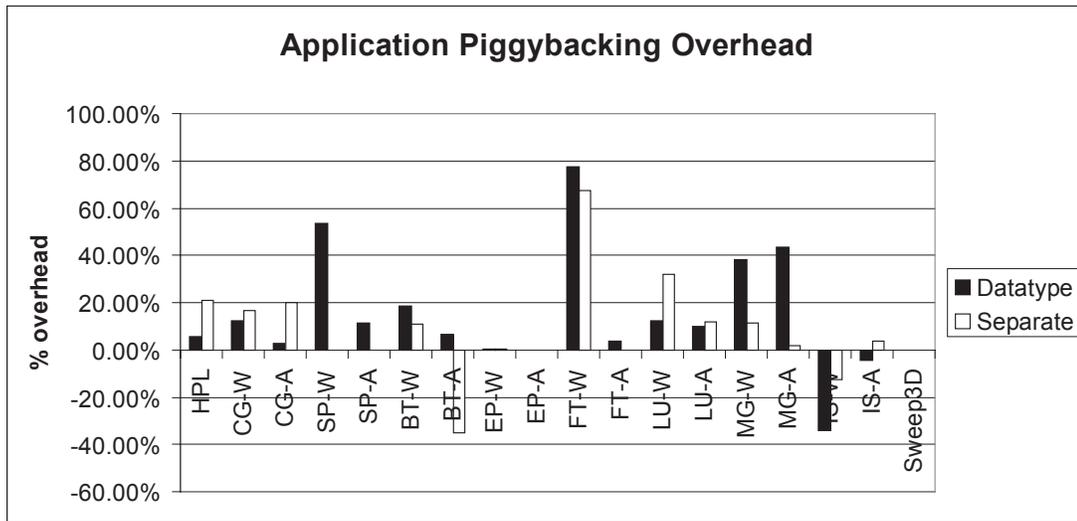


Figure 2.52: Application piggybacking overhead on 16 processors, Velocity 1

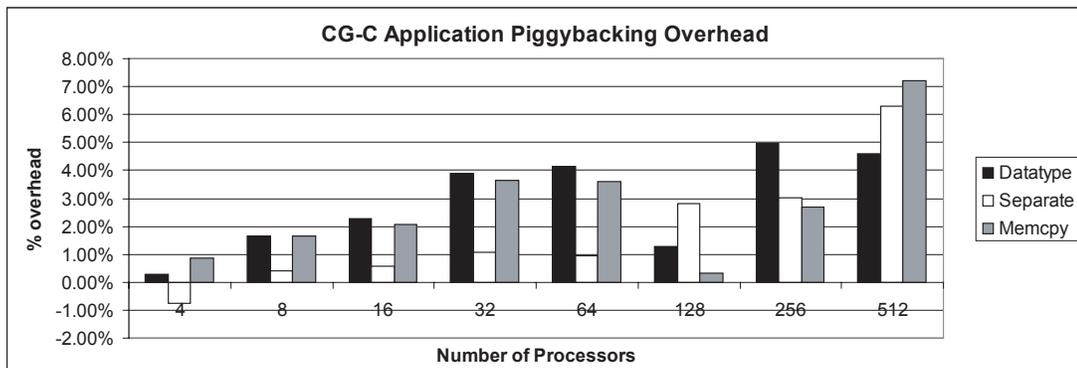


Figure 2.53: Piggybacking overhead for CG-C on 4-512 processors, Lemieux

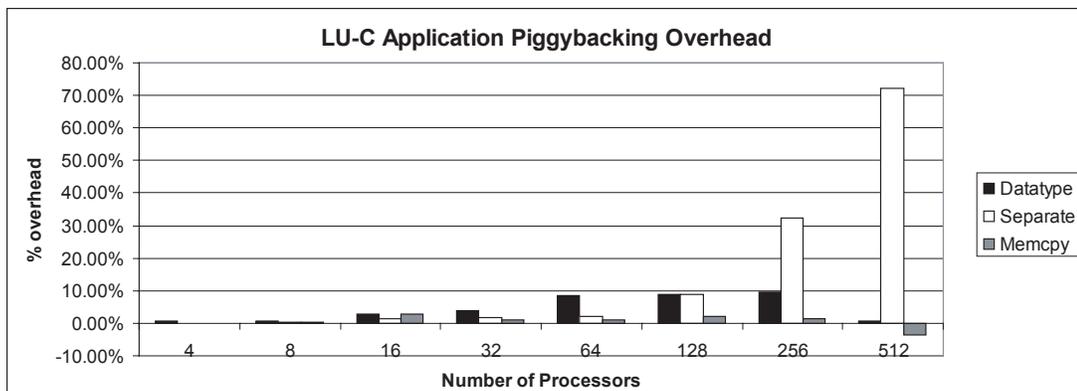


Figure 2.54: Piggybacking overhead for LU-C on 4-512 processors, Lemieux

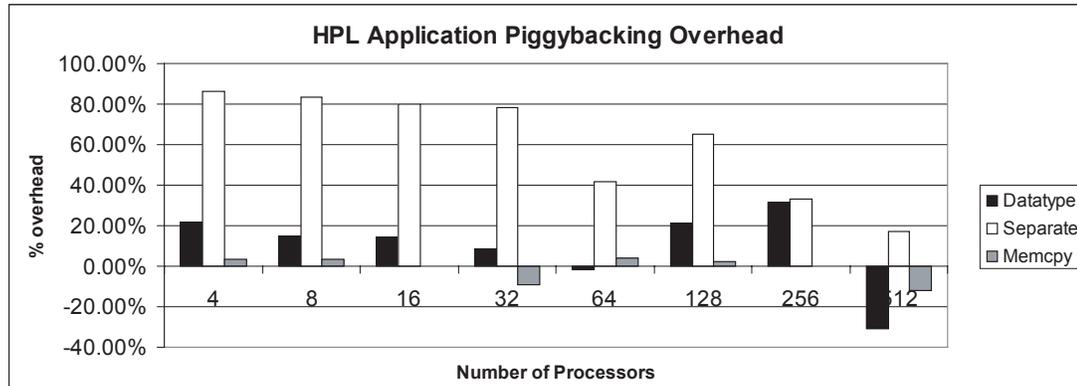


Figure 2.55: Piggybacking overhead for HPL on 4-512 processors, Lemieux processors in input classes W and A. The former has overheads of 70% and the latter has almost no overhead.

- Running the same application with the same input size on different numbers of processors can have a dramatic effect. In particular, the piggybacking overhead for SP on input class W with datatype piggybacking is 5% on 4 processors and 50% on 16 processors. Another example can be seen in the LU benchmark running on 4 - 512 Lemieux on input class C (Figure 2.54) with all three piggybacking mechanisms. Both datatype and memcpy have low overhead for few processors, which rises for 64-256 processors and falls back to 0% for 512 processors. Meanwhile, separate piggybacking induces small overheads with few processors, which rises steeply as LU is run on 128 and more processors, climbing to to >70% for the 512 processor run.

**2.5.10.3.4 Discussion** Given the above lessons, the bottom line is that piggybacking above the MPI interface causes non-trivial overheads to real applications, both in the sense of being difficult to analyze and being potentially very large. The piggybacking experiments for point-to-point messages show that although there exists an intuitive analytic model for the expected bandwidth and latency

overheads of separate and memcpy mechanisms, this model is not a good predictor of bandwidth overhead and can frequently be wrong regarding latency overhead. Furthermore, it suggests no way to tune the model to a given architecture (for example by identifying the memcpy/separate overhead crossover point) without actually running the latency and bandwidth tests.

The effect on the performance of real applications is even more difficult to understand, with modest variations in application parameters causing overhead behavior to change significantly. Although attempts were made to create a set of microbenchmarks, runtime tests and heuristics that might be able to reliably suggest the best piggybacking mechanism to use for a given run of a given application, such a technique has proven elusive, with a variety of different prediction techniques showing poor accuracy.

Fortunately, given that RROMP currently utilizes only three types of piggybacking, it is possible to simply try all three out at runtime and pick the one that performs the best. The protocol switch-over could be implemented using a sync-and-stop type of protocol that temporarily blocks all communication until it could be made clear what messages were sent before the switch-over point. These messages would be received using the old piggybacking mechanism and subsequent messages using the new mechanism. Determining the performance impact of a given piggybacking mechanism can be done by measuring the application's cache performance, the average amount time each process is blocked at receives, or other standard performance profiling techniques.

# Chapter 3

## Parallel Checkpointing - Shared Memory

### 3.1 Background

The idea behind the shared memory programming model is that the tasks of an application have direct access to each other's memories, allowing them to interact with each other without explicit message send and receive operations. The main advantage of shared memory over message passing is in ease of use. Since no explicit communication is required to exchange information between different threads, shared memory makes it possible to initially implement the application in a sequential fashion and then iteratively optimize it to run with multiple threads.

The primitive operations of any shared memory API are reads and writes to shared memory locations. These operations alone are generally sufficient for applications' record-keeping and communication purposes. However, the fact that multiple threads are simultaneously accessing the same address space poses a problem: without some access coordination, the reads and writes of different threads will interfere with each other in complex ways, making shared memory programming quite difficult. As such, most shared memory APIs provide a number of additional primitives for thread coordination. These include the atomic update operations (such as compare&swap, test&set and test&increment), locks, semaphores, critical sections, critical regions and monitors, with different APIs providing different sets of coordination primitives. In addition to coordination, many APIs provide additional useful functionality. One classic example is data distribution primitives such as those in High Performance Fortran [14] where the application can specify how different arrays are to be physically laid out in the memories of the different

processors running the application. Another example is parallel for loops, such as those of OpenMP [15] and UPC [24] where different iterations of the same loop are automatically assigned for execution to different threads.

Shared memory can be found in the full spectrum of real-world systems, ranging from low-power embedded processors to 512-processor supercomputers such as the SGI Altix. Its relatively simple programming model has made it a popular choice for both application programmers and operating system implementors. Its many implementations range from hardware consistency protocols to single-processor software thread schedulers to software shared memory implemented on top of distributed memory hardware. Operating Systems feature a variety of shared memory APIs, including Posix Threads [20] and Win32 threads [66]. Java Threads [156] and C# Threads [44] bring shared memory to a wide audience of programmers working with these languages. OpenMP [15], UPC [24], Co-Array Fortran [155] and Global Arrays [154] are among the many shared memory APIs commonly used in high performance computing.

Parallel rollback restart for shared memory applications differs from the problem for message passing APIs because memory becomes an entity of its own, separate from the threads that make up the application. While each thread may have some "private" memory that only it can access, the common shared address space is independent of all threads. Therefore, if one thread is stopped and later restarted, it is imperative to maintain for it the illusion that shared memory did not change in its absence whereas in reality it may have changed substantially as a result of application activity. The same is true when all threads are rolled back because on restart they must be provided with an image of shared memory that is consistent with the state of shared memory at the time of the checkpoint.

### 3.1.1 Shared Memory Models

Any discussion of rollback restart for shared memory must focus on the various models of shared memory and the important implications they have for any rollback restart solution. A memory model is simply a description of how read and write operations executed by the same and different threads behave relative to each other. The sequential memory model is simple: a read to a given variable must return the value written by the most recent write to this variable. The simplest shared memory model, sequential consistency [130], extends this simple model to the multi-threaded case. The intuition behind sequential consistency is that the execution of the application must correspond to some interleaving of operations on different threads executing on the same processor and accessing the same memory. More formally, the results of all reads performed by the application's threads must correspond to some total order on all read and write operations such that a given read of a variable returns the value of the write to this variable that was most recent according to the total order.

Sequential consistency provides programmers with an intuitive memory model, allowing them to pretend that all of the application's threads are in fact running on a sequential system. It is implemented in most single-processor shared memory libraries and even in some cluster implementations such as Kerrighed [143]. However, enforcing sequential consistency on a multi-processor system (i) requires a significant amount of stalling and additional communication and (ii) prevents the hardware and the compiler from employing common optimizations such as instruction reordering and caches that regular sequential codes benefit from. This results in reduced performance for systems that guarantee sequential consistency, a phenomenon that has been verified experimentally [161] [94].

In light of the cost of providing sequential consistency, a number of additional weaker memory models have been developed, including processor consistency [97], weak ordering [73] [74], release consistency [95], location consistency [91], entry consistency [42], total store ordering [21] [191], partial store ordering [21] [191], the Digital Equipment Alpha memory model [192] and the PowerPC memory model [141], among others. These models are designed to allow for additional hardware and compiler optimizations, such as instruction reordering, the use of non-FIFO networks and having multiple processors simultaneously write to the same variable that is stored in their private caches. The result from application's perspective are violations of sequential consistency such as writes performed by other thread not being visible to other threads and concurrent writes by different threads appearing to execute in different order from the perspective of different threads.

For a concrete example, consider the weak ordering memory model. It separates memory accesses into **data** and **synchronization** accesses. No ordering provided for **data** accesses: the shared memory implementation is allowed to reorder operations on different variables. **data** writes are guaranteed to be atomic: no thread may see the partial result of a **data** write. **synchronization** accesses are provided with much stronger guarantees. First, any interactions between **synchronization** accesses are guaranteed to be sequentially consistent. Second, all **data** accesses that precede a given **synchronization** access in the program's source code must complete before this access may begin. Similarly, **data** accesses that follow a given **synchronization** access in the program's source code may not begin until this access has completed.

The intuition behind weak ordering is that most of each thread's memory ac-

cesses will be to variables that no other thread is touching at the same time. In this case it can use the highly optimized **data** accesses, which in the absence of multi-thread data races behave as if they were sequentially consistent. However, threads occasionally need to synchronize their executions in order to ensure that **data** accesses do not race with each other. **synchronization** accesses can be used for this task. Although their guarantee of sequential consistency makes **synchronization** accesses slower, their relative rarity means that the application's performance is almost the same as if all of its accesses are the fast **data** accesses.

While weak ordering already provides a relatively limited set of guarantees, it can be further weakened to produce release consistency. The idea behind release consistency is that most synchronizations used by the application correspond to the acquisition or release of some resource. The weak ordering rule that the execution of a **synchronization** access may not be reordered relative to a **data** access conservatively accounts for both possibilities but is too strong in practice since the programmer typically knows whether a given **synchronization** access acquires or releases a resource. As such, release consistency provides special access types for both kinds of synchronizations, breaking up the notion of **synchronization** accesses into separate **acquire** and **release** accesses. Specifically, the execution of a **data** access that appears after an **acquire** in the program source code may not begin execution until the **acquire** has completed. Conversely, the execution of a **data** access that appears before a **release** in the program source code must finish executing before the **release** may begin execution. Because of its looser semantics, release consistency has been popular in software implementations of shared memory such as Treadmarks [36], KAIST [125] and SCASH [104].

The choice of the guarantees provided by a memory model significantly in-

fluences the implementation of the shared memory system and the complexity of writing applications for such a system. While there have been efforts to describe memory models in a more programmer-centric fashion [28], these models are still far from transparent to the programmer. However, given the significant performance advantages afforded by relaxed consistency models, especially for scalable shared memory machines (the Sun Fire 25K scales to 72 processors and the SGI Altix to 512), the benefits of relaxed memory models are generally considered to be well worth the price.

## 3.2 Prior Work

There has been a variety of work on developing rollback restart solutions for shared memory. The space of possible solutions is strongly influenced by the way shared memory is commonly implemented, as shown in Figure 3.1. Applications are written to work with some shared memory API and memory models, where one API may support one or more memory models while each memory model may be supported by one or more APIs (while a memory model defines the behavior of reads and writes, a full API provides additional functionality such as synchronization functions and data parallel primitives). Each API is implemented by one or more shared memory implementations in hardware, software or a combination of both. Each implementation may use one or more consistency protocols and a given protocol may be used by one or more implementations. Shared memory implementations run on one or more network fabrics that at a low level provide a message passing model of communication that must be used by the implementation to provide the application with the abstraction of shared memory.

Since all shared memory implementations are ultimately based on message

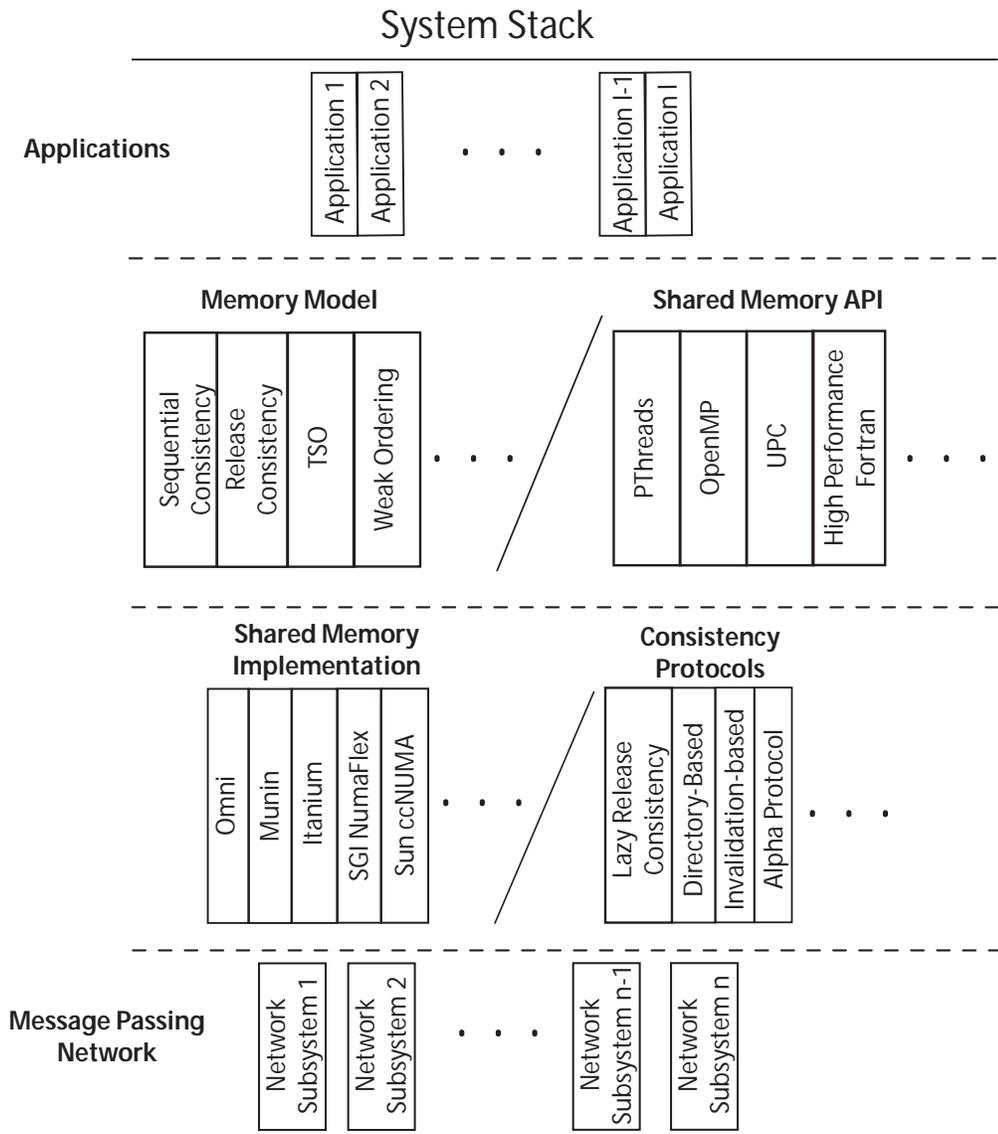


Figure 3.1: Space of shared memory rollback restart solutions

passing, it is theoretically possible to use the techniques described in Chapter 2 to provide rollback restart for any shared memory implementation. However, in practice this can be unnecessarily expensive since shared memory implementations send many small messages, many of which are not relevant for restart. Moreover, they perform many non-deterministic actions (every read may be a non-deterministic event), further complicating this approach and reducing its efficiency. As such, typical solutions work at the level of the shared memory implementation, tailoring themselves to the details of a particular implementation, memory model or consistency protocol.

The wide variety of shared memory implementations and protocols means that the work on rollback restart of shared memory applications has been fragmented. While there exist rollback restart solutions that support a variety of shared memory implementations, the very large size of this space has meant that even at their broadest, existing rollback restart solutions have a fairly narrow scope of applicability. This is in contrast to rollback restart for message passing where the simplicity and uniformity of this communication model has made it possible for a single message passing rollback restart protocol to be applicable to most message passing implementations (ignoring efficiency). The result is that the high heterogeneity of shared memory rollback restart solutions has severely limited their availability in the real world, to a point where today, after decades of research, real implementations are still difficult to find.

The fact that shared memory protocols have the same types of inter-processor interactions as their message passing counterparts suggests that we can organize them into a similar taxonomy. This presentation divides shared memory rollback restart protocols into the following types: coordinated, uncoordinated, quasi-

synchronous and message logging. Protocols in each of these families closely resemble their message passing counterparts but focus more tightly on the internal details of their target protocols. Given the number of known protocols, it is not possible to survey the entire field of possible solutions. As such, the following sections merely present a number examples, describing the details of each rollback restart solution and the protocol(s) that it is tailored to.

### 3.2.1 Coordinated Checkpointing

SafetyNet [194] is a framework for adding checkpointing to hardware multiprocessors for the purpose of surviving transient faults. It focuses on the sequential consistency memory models and augments existing protocols with checkpointing functionality ([194] shows how to do it for a particular directory-based protocol). SafetyNet relies on the availability of some sort of logical time being maintained by the system's processors such that no message can arrive at an earlier logical time than the time when it was sent (loosely synchronized clocks with drift smaller than minimum message latency satisfy this requirement). All processors checkpoint themselves at the same logical time, once every 5,000-1,000,000 processor cycles (thousands of times per second). The processor's registers are checkpointed directly to a nearby checkpoint buffer while its cache and memory are saved lazily in a copy-on-write fashion. In order to make sure that the ownership of each memory block is known at the time of a checkpoint and is not in some intermediate state, a given checkpoint is not committed until all processors have determined that all their memory block ownership requests issued before the checkpoint have been completed. SafetyNet shows very low overheads because of its relatively rare (on a processor time scale) checkpoints and the fact that they log only .1% of all

cache accesses. However, it is also quite limited in the severity of the faults that it can deal with. In particular, it cannot roll back from with processor failures or cache corruptions due to soft errors.

Revive [175] is another example of a hardware coordinated checkpointing protocol, except that it uses blocking coordination instead of the non-blocking protocol used by SafetyNet. Revive targets off-the-shelf processors with caches and modifies the networking components of common hardware shared memory implementations, including network cards, directory controllers and memory controllers. To record a checkpoint all processors synchronize using a 2-phase commit protocol and flush their pipelines and caches, causing the main memories to contain a consistent state of the shared address space. Since main memories are still vulnerable to failure, their state is encoded using a single-bit parity error correcting code, with extra memory modules dedicated to holding the parity bits of the other modules. The memory controllers are modified to ensure that the parity module is always upto date with any changes to the other memory modules. Whenever a memory location is written to for the first time after a checkpoint, its original value is copied out into a special log in a copy-on-write fashion before the write is allowed to proceed. This is to ensure that the state of the memory at the time of the checkpoint is not lost. On restart the state of the shared address space is recreated from the current memory state and the overwritten values stored in the log. The result is that Revive has an time overhead of 6% and memory overhead of 14% while taking 10 checkpoints per second. Under their failure probability assumptions in [175], this comes out to 99.999% availability.

Coordinated checkpointing is extended to software shared memory by [56], which presents a protocol closely related to sync-and-stop where processors check-

point themselves inside application barriers. Because at barrier-time all shared memory state is consistent and there is no in-flight communication, it is possible to simply save the state of all pages and all directories in a checkpoint and directly restore them on restart. However, the fact that the application's own barriers are used for synchronization means that no additional coordination cost is incurred. The paper also presents an experimental evaluation of this protocol on a 56 node Intel Paragon, showing that the cost of saving application data to disk is the primary cost of checkpointing and that optimizations like incremental checkpointing and copy-on-write can significantly reduce this cost. [126] extends blocking checkpointing to the specific case of software implementations of lazy release consistency (Treadmarks [36]), where they improve upon the baseline performance of incremental checkpointing by applying it to Treadmarks' internal data structures.

[71] presents a software-based checkpointer for SMP systems. In particular, their approach works at the user-level, above the POSIX Threads API [20] (PThreads). As such, they do not have direct control over the implementation of the API and in particular over the Operating System's kernel threads. Their checkpointing protocol (which bears relation to the generic protocol presented in Section 3.4) is initiated by a single master thread that sends a signal to all other threads. Since PThreads does not define what happens when a thread that is blocked on a mutex receives a signal, it is possible for such blocked threads to never be informed that a checkpoint has begun. As such, the master thread, along with any threads that have begun their checkpoint, release all their mutexes so that blocked threads may acquire them and be informed that a checkpoint has begun. In order to ensure that threads that have been released on this fashion do not continue executing application code, potentially corrupting application state, the

master thread sets the `prevent_locks` flag in order to inform forcefully released threads that they should not continue executing. When all threads have been informed of the checkpoint, they block and the master thread saves the state of the process to disk. It then informs all the other threads, which reacquire their released mutexes, and resume executing application code, which may involve re-blocking on mutexes that they were forcefully released from during the checkpoint.

The solution in [71] has the advantage of high portability, since it works above the PThreads implementation rather than inside of it. Furthermore, their experiments show relatively low overheads for the single platform they have tried. However, they go beyond the PThreads specification in a number of ways, which requires them to make a number of assumptions about the internal workings of PThreads that may not always hold in practice, with no suggestions for what to do if those assumptions do not hold. The biggest problem is the fact that at checkpoint time they save all application state without differentiating whether it belongs to the application or the PThreads implementation. This makes restart difficult since the PThreads implementation may have pre-restart information that may not remain valid on restart. However, this work presents a high water mark in solution generality in prior work on shared memory checkpointing.

BLCR [75] takes a different approach toward checkpointing PThreads applications by embedding this functionality inside the Linux kernel. While this approach sacrifices portability, kernel-level solutions have much more power over the PThreads implementation than their user-level counterparts like [71] and  $C^3$ . In BLCR's checkpointing protocol the master thread sends a signal to each application thread. When this signal is delivered to a thread, the signal handler makes a system call, informs the master thread and blocks the thread at a barrier. When

the master thread has been informed that all threads are waiting for it, it releases them from the barrier and all threads participate in saving the application's state. A checkpoint is terminated via another barrier, after which point all threads return from the system call and resume executing application code. The relative simplicity of working at the kernel level has helped to make BLCR more functional but has limited its portability, meaning that BLCR has required updates to work with new versions of the Linux kernel (2.4 to 2.6) and new instruction sets (x86 to x86\_64 and IA-64).

### **3.2.2 Uncoordinated Checkpointing**

In uncoordinated checkpointing individual threads are allowed to independently checkpoint their own state without any additional synchronization or communication between them. While this minimizes the regular-execution time overhead of checkpointing, on restart it can cause threads to roll back far into the past via the domino effect [179], causing both a time overhead and a space overhead for additional checkpoint storage. The extent of the domino effect depends directly on the frequency of communication in a given application. Since shared memory applications tend to feature significant amounts of communication using small messages, uncoordinated protocols appear to be a poor choice for the shared memory checkpointing problem. As such, thus far they have not been a popular research direction.

### **3.2.3 Quasi-Synchronous Checkpointing**

The approach in [216] focuses on providing fault tolerance for distributed memory implementations of shared memory. The primary feature of such implementations

is that each processor has its own memory (not just cache) that is not accessible by other processors. In particular [216] extends the invalidation-based protocols from [134] to support checkpointing. The protocol in question uses the disk to hold the contents of the entire application address space, with processors holding copies of some of these pages in their memories. Multiple processors may have a read-only copy of a page in their memory but if a process wishes to write to this page it must first invalidate all the read-only copies by sending messages to their host processors. If later on some processor wants to read from or write to this page, it must ask for the modified copy to be sent over and if it is trying to read it, change the page's status to read-only.

In [216] a processor checkpoints by (i) saving its state to disk and (ii) flushing all the dirty pages modified by the processor since the last checkpoint to disk. On restart all of its memory pages are initially invalid. It can then reacquire any pages it needs by loading them from other processors or the disk whenever the application tries to read from or write to them via the same shared memory mechanisms that are used for normal read and write requests. One complication is that allowing processors to checkpoint independently can cause cascading rollbacks, just like the domino-effect discussed in Section 2.3.2. For example, let processor  $p$  take a checkpoint and then write to some page. Let processor  $q$  then read this page (by asking  $p$  to send it this page and transitioning the page to read-only state) and take a checkpoint. If  $p$  fails and needs to be rolled back, processor  $q$ 's read behaves essentially as an early message, forcing it to also roll back. These rollbacks may force the system to roll back to the start of the application's execution. The solution proposed by [216] is a quasi-synchronous protocol (closely related to the message passing protocols in Section 2.3.3) where a process is forced to take a

checkpoint when another process asks it for a page that it has modified since its last checkpoint. Because this scheme has the potential to take large numbers of forced checkpoints, [216] also presents a scheme for efficient incremental checkpointing.

### 3.2.4 Message Logging

[200] applies sender-based message logging to an implementation of the Home-based Lazy Release Consistency (HLCR) protocol [117], which supports the release consistency memory model. Each page is associated with some home processor that maintains the most recent version of the page. Each processor maintains a local time = the number of synchronization operations it has executed thus far. The time of each event is represented as a vector clock of local processor times that precede the event and events are ordered via their vector times into a partial **happens-before** order. When a processor  $p$  executes an **acquire** operation it sends a request to the owner processes of each of its pages to determine whether they know of a version of any of these pages that is more recent than the version held by  $p$ . If so, the newest version is sent to  $p$  and  $p$ 's vector clock is updated to be no earlier than the vector clocks of all the incoming pages. When  $p$  executes a **release** operation, for any page that it has modified since its last synchronization operation,  $p$  gives this page the timestamp of the **release** operation and sends its modifications (in the form of a diff) to the page's home processor.

The protocol presented in [200] allows HLCR to tolerate the failure of a single processor and works as follows. Processors checkpoint themselves independently. Each processor's checkpoint contains (i) the processor's internal state, including any non-shared memory and (ii) the data of any pages that it is home to. Because these checkpoints are uncoordinated, they are vulnerable to cascading rollbacks

from the domino effect. [200] deals with this via sender-based message logging. As discussed in Section 2.4, a message logging protocol must do three things: (i) record the outcomes of all non-deterministic events, (ii) record the data of any communication from processors that did not roll back to the one that did (will be resent on restart) and (iii) suppress any communication from the rolled back processor to the others until it has finished restarting. Non-determinism appears in HLRC in the behavior of locks where after a given processor has released a lock, the choice of the processor that will acquire it next is non-deterministic. As such, whenever a processor acquires a lock, both it and the lock's releaser record this event in logs that they keep in their volatile memories. The only forms of communication in HLRC is processors sending page modifications during `release` operations and home nodes sending page copies to nodes that are performing `acquire` operations. As such, both of these communications are logged at their senders. Writer processors log the diffs that they send to home nodes and home nodes log old versions of their pages. This data is also kept in volatile memory.

When some processor rolls back, it reloads its state and the states of all of its pages from its checkpoint. It then receives from other processors (i) the details of when it acquired its locks, (ii) the data of any pages that it received from other processors in the course of its execution and (iii) the updates sent to it by other processors when they modified the pages that it is home to. It uses this information to locally recompute its pre-rollback state, including the state of the pages that it was home to, without doing any additional communication. When it has finished this restart process, it can once again participate in computation.

A logging scheme like this has two primary overheads: the cost of checkpointing and the cost of logging. [200] tries to reduce these costs via a mechanism for

trimming from the log old entries that are no longer needed and a garbage collection mechanism that deletes old unneeded checkpoints. Their experiments show these mechanisms to be effective but also reveal an interesting overhead of this solution. Applications with significant amounts of synchronization where there is also a load imbalance in terms of the number of pages that a given processor is home to have a large overhead with this protocol. The reason is that the home node of a given page needs to log all changes to this page, meaning that some nodes will be slowed down by logging more than others. In applications that synchronize frequently this will cause all processors to slow down, causing a significant degradation in performance.

[150] presents single-failure causal message logging protocol for the DiSOM [102] distributed shared memory system. DiSOM implements the entry consistency [42] memory model, where applications explicitly associate their data objects to synchronization objects. The application may only access a data object once it has acquired its respective synchronization object, which allows the shared memory implementation to only keep the state of data objects consistent at the acquisition and release points of their respective synchronization objects. These are also the only points in time when non-deterministic behavior can occur.

The protocol in [150] works by having each processor independently record its checkpoints. During regular execution processes record (i) the different versions of each data object produced by each thread and (ii) information about the order of acquire and release operations. This information is sent to neighboring processors as part of DiSOM's regular consistency protocol and stored in their volatile memories. When some processor fails, all other processors send to it all the data object versions it received during its pre-failure execution and the outcomes of all

of its acquire-release non-deterministic decisions. It is then able to deterministically recompute its failure-time state. Since each piece of information required to recreate the state of each processor is held by a single neighbor processor in its volatile memory, this protocol can survive the failure of only one processor at a given point in time.

### 3.2.5 Miscellaneous

[31] discusses an extension of the Cache-Aided Rollback Error Recovery (CARER) technique [115]. CARER maintains exactly one checkpoint at all times. Processor state is saved by using a set of backup registers into which the values of regular registers are copied. Memory is checkpointed by flushing all dirty cache lines to main memory. After this has been done main memory corresponds to its state at checkpoint time and any writes performed by the processor are kept in dirty cache lines, which are not flushed to main memory until the next checkpoint. In the shared memory context [31] proposes three checkpointing protocols for coherence protocols that work with (i) processors that have caches, (ii) are connected by a bus and (iii) are connected to a single common main memory. The first protocol has all processors checkpoint at the same time and when one processor rolls back, so does every other (i.e. this is a coordinated protocol). The second protocol tracks the read/write interactions between processors and when a processor takes a checkpoint, it only forces other processors to checkpoint if it has participated in interactions since its last checkpoint and only processors that have interacted with others will need to checkpoint (i.e. it's a quasi-synchronous protocol). The same logic is used to decide which processors will roll back. The third protocol forces processors to checkpoint whenever they participate in interactions with

other processors (again, a quasi-synchronous protocol). The advantage is that when a single processor wants to restart, no other processor is forced to restart as well. While they provide no experimental data, their modeling suggests that these algorithms can significantly degrade system performance.

Transactional memory is a variant of shared memory, where certain pieces of code are identified as transactions. Code in these transactions can access a shared address space and the underlying runtime system is responsible for guaranteeing the illusion that these transactions are executed in some serial order. Although this is trivial to guarantee on sequential systems, on parallel systems this is typically done by optimistically running multiple transactions at the same time and dynamically detecting if one transaction writes to a variable that is accessed by a concurrently executing transaction. If a conflict is detected where two transactions cannot be sequentially ordered relative to each other, one is rolled back and retried. Research on this key mechanism of rollback restart has studied for decades in the context of database transactions [177] and very recently in research on transactional memory models [105].

### **3.3 Availability of Rollback Restart**

Despite the great variety of rollback restart protocols for shared memory systems, they are almost uniformly consistent on one point: no protocol applies to more than a small fraction of the shared memory solution space. As shown in Figure 3.1 shared memory systems are created via a layering of APIs, memory models, shared memory implementations, consistency protocols and network interfaces. Each protocol focuses on a fraction of this space: a particular memory model, a type of network interface or a type of consistency protocol. As such, no protocol can hope

to provide rollback restart functionality for even the majority of applications.

While implementation portability concerns have severely limited the availability of rollback restart for message passing applications, the protocol portability problem has made rollback restart virtually unavailable in real systems. Today, the only shared memory systems that have rollback restart functionality are software threading packages or software shared memory implementations that use their own custom-designed protocols. The result is that today rollback restart for shared memory remains an active area of research with little hope of advancing to popular use on real systems in the near future.

[71] presents a rare solution that presents a protocol that is independent of the underlying memory model or consistency protocol. Unlike other approaches, it works above the PThreads API, providing checkpointing for any application running on top of many implementations of this API. Although [71] makes a number of assumptions about the underlying implementation details and although it focuses exclusively on the PThreads, it represents a high water mark in generality for rollback restart solutions that points the way towards making such solutions widely available to the general public.

It is, of course, possible to go further. One option would be to take a high-performance implementation of shared memory that works on a wide variety of platforms and supports a variety of shared memory APIs and augment it with rollback restart functionality. While appealing in principle, the fact is that there is no one shared memory implementation that both works on and offers high performance for the majority of shared memory systems and the majority of shared memory applications. While there have been efforts to create such an implementation, including Omni [16] and Nanos [39], success has been limited by the wide

variety of hardware and software involved. Indeed, the technical and organizational challenges are formidable enough that it is doubtful that such an implementation will be created in the intermediate future.

Another option would be to go beyond [71] and create an application-level rollback restart protocol that would work with any shared memory implementation, consistency protocol and memory model (not just PThreads). Such a protocol could then be applied to any shared memory API to create a full rollback restart solution for shared memory applications. As an application-level approach, it would require one round of effort to apply this protocol for each shared memory API of interest. This is in contrast to lower-level solutions that would require a round of implementation effort for each shared memory implementation, consistency protocol and/or memory model, which are far more numerous than shared memory APIs. While a protocol of this type would not solve the problem completely, it would at least eliminate the major problem of restart protocol non-portability and significantly reduce the barriers that currently impede the creation of rollback restart solutions that are useful for the vast majority of real applications.

This chapter presents such a protocol. It is a blocking coordinated checkpointing protocol that synchronizes all threads and has them record the full state of the application. It works above the shared memory API and therefore does not make any assumptions about the underlying implementation. This lack of control over implementation internals makes it more difficult to coordinate thread checkpoints while avoiding deadlock, resulting in a relatively more complicated protocol. However, the resulting solution can be adapted to most shared memory APIs and makes very few assumptions about the underlying implementation details. In order to experimentally show the applicability of this protocol to real shared memory APIs

and implementations, it is applied to OpenMP [15], a popular shared memory API. The implementation of this protocol on top of OpenMP is experimentally evaluated on four different hardware/software platforms using three sets of benchmarks and shows generally low overheads despite having no knowledge of the underlying details of these platforms.

## 3.4 Portable Checkpointing Protocol

### 3.4.1 Approach

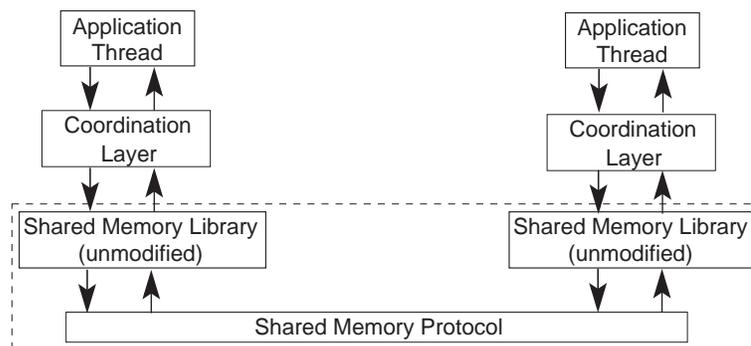


Figure 3.2: Overview checkpointing approach

Figure 3.2 describes the approach. The checkpointer sits as a thin layer between the application and the underlying shared memory system. It intercepts all the function calls that the application may issue (thread creation, resource acquisition, etc.) but not the reads and writes (for performance reasons). The job of the checkpointing layer is to coordinate the state saving activities of all threads such that the checkpoint saved on stable storage can be used to restart the application when needed. It is assumed that there exists some way of recording the state of the application. In particular, there exists a function `Save_State`, which, if called by every thread, is guaranteed to record the private and shared state of all threads to stable storage (the portion of system state saved by each thread is

undefined). The state of the shared memory library and the underlying shared memory protocol is assumed to be not saved and is instead handled by the shared memory checkpointer.

The basic protocol used to coordinate the state saving is shown in Figure 3.3 and is executed by every thread:

```
Global_Barrier
Save_State
Global_Barrier
```

Figure 3.3: Blocking Checkpointing Protocol

The two barriers ensure that the application is not running while the checkpoint is being recorded. Furthermore, the first barrier ensures that any writes to shared data that were initiated before the checkpoint, will have completed by the time the first barrier returns. This makes this protocol applicable to both sequential consistency as well as any relaxed memory model model, since in virtually any memory model barriers force memory to be consistent at that point in time.

### 3.4.2 Synchronization and the Protocol

While it is easy to see how the basic protocol in Figure 3.3 can successfully save application state, the fact that it is blocking presents difficulties if the application uses synchronization to coordinate its own activities. In particular, suppose that the API in question allows the application to acquire and release mutually exclusive resources. The problem arises when one thread tries to checkpoint while holding a resource that another thread is waiting to acquire a resource. This can result in deadlock, as shown in Figure 3.4(a) where thread 0 blocks at the first global

barrier within the checkpoint protocol while holding a resource while at the same time thread 1 blocks trying to acquire that same resource.

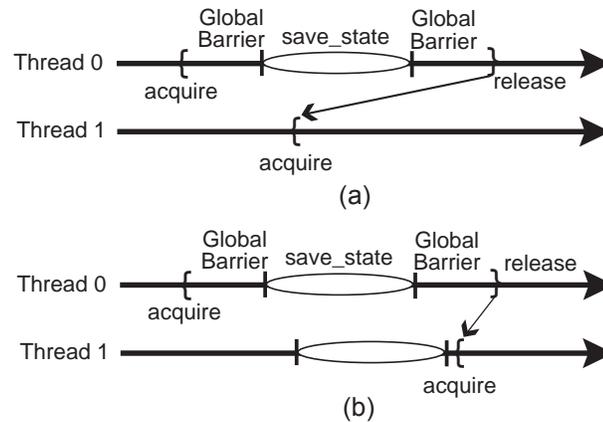


Figure 3.4: Deadlock example

Since a checkpoint has begun but has resulted in a deadlock, two design choices present themselves. One option is to abort the checkpoint and try taking it at a later time. The problems with this approach are that it wastes the time already spent on the aborted checkpoint and it does not provide a guarantee that a checkpoint will even be taken. Another option is to wake thread 1 up and inform it that it must participate in this checkpoint. Thread 1 would then take a checkpoint and would only resume acquiring the resource after the checkpoint was completed, transforming Figure 3.4(a) into Figure 3.4(b). The problem with this solution is that it may force threads to take a checkpoint at sub-optimal locations in their code (i.e. those that do not have a minimal amount of state). Given the tradeoffs inherent in both options, this protocol chooses the second option of checkpoint forcing because sub-optimal checkpointing is better than the possibility of never checkpointing at all.

### 3.4.3 Refining the Basic Protocol

The checkpoint forcing approach requires threads blocked on resource acquisition operations to be woken up and forced to take a checkpoint. However, without knowledge of the details of the underlying shared memory implementation the only mechanism to wake up a thread that is blocked on a resource that is generic among all shared memory APIs is to actually give it that resource. This is the approach taken in this protocol:

- Before checkpoint begins: ensure that all threads are awake by releasing all resources.
- Before checkpoint completes: each thread reacquires its released resources.
- Immediately after checkpoint completes: put each thread that that was forcefully awoken back into a blocked state by having it try to reacquire the resource it was blocked on.

When a thread acquires a resource it must determine whether it woke up because the resource was released by application source code or by the checkpointing protocol. As such, the protocol sets the shared counter `checkpoint_initiated` and shared flags `threads_awoken` before it starts releasing resources. When a thread acquires a resource, it checks the above variables to determine whether it was awoken by the protocol or the application. If the former, it releases any resources it just acquired and immediately takes a checkpoint. If the latter, it proceeds with its regular computation.

This basic template means different things for different type of resources.

- Locks and critical sections:

- Before checkpoint begins: the thread holding each such resource must release it;
  - Before checkpoint completes: each owner thread must reacquire its resources;
  - After checkpoint completes: each forcefully awoken thread tries to reacquire the resource it was blocked on.
- Barriers:
    - Before checkpoint begins: all threads that are not waiting at a barrier must execute a barrier in order to release the threads blocked at a barrier;
    - After checkpoint completes: each forcefully awoken threads again blocks at a barrier.
- Semaphores:
    - Before checkpoint begins: the semaphore counter must be decremented  $n$  times where  $n > 0$  and  $n \leq$  semaphore counter at the time of the checkpoint;
    - Before checkpoint completes: the semaphore's counter is  $n$  less that it was before the checkpoint began, so it must be waited on  $n$  times to restore the state of the counter to its pre-checkpoint value;
    - After checkpoint completes: each forcefully awoken thread tries to wait on the semaphore again.
- Condition variables:

- Before checkpoint begins: each condition variable must be repeatedly notified for as long as it is not known that all threads have begun their checkpoints;
- After checkpoint completes: each forcefully awoken thread tries to wait on the condition variable again.

Other types of synchronization can also be supported by this framework but their incredible variety prevents us from addressing them all here. Since this protocol uses the underlying shared memory implementation's acquire and release operations in order to coordinate its checkpoints, it is an example of the Replay methodology from the 4R taxonomy of checkpointing techniques.

Figure 3.5 shows the complete protocol framework<sup>1</sup>. Function `perform_checkpoint` may be called at any time in any thread's execution when the thread has control. In other words, it may not be called while the thread is executing inside the shared memory library and/or blocked on some resource, since the state of the thread may be undefined at that point in time.

When performing a checkpoint, a thread first atomically increments `checkpoint_initiated` in order to inform other threads that a checkpoint has begun (it contains the number of threads that have already begun their checkpoint). It then releases all the resources it holds, which may include locks, critical sections, semaphores, etc. by calling `forcerelease_xxx`, where `xxx` corresponds to their respective resource types. It helps to release any threads blocked at a barrier by calling `forcerelease_barrier`. After performing the first checkpoint barrier, all the threads participate in saving the state of the application. Before

---

<sup>1</sup>For clarity, the code executed on restart is not shown. See Section 3.8.3.5 for more details.

```
shared int checkpoint_initiated = 0;

void perform_checkpoint(bool at_barrier) {
    Atomic checkpoint_initiated++;
    foreach r (resources_held_by_current_thread())
        forcerelease_xxx(r);
    if (!at_barrier) forcerelease_barrier();

    Global_Barrier();

    Save_State();

    checkpoint_initiated = 0;
    foreach r (resources_held_by_current_thread())
        reacquire_xxx(r);
    if (!at_barrier) reset_barrier();

    Global_Barrier();
}
```

Figure 3.5: Checkpointing Protocol Framework

performing the second checkpoint barrier, each thread reacquires all of the resources it held before the checkpoint was initiated by calling `reacquire_xxx` on each `reacquire_xxx` on each such resource and calling `reset_barrier` to reset the checkpoint notification flags used by barriers. The parameter `at_barrier` is set to true if `perform_checkpoint` was forcefully called while an application was waiting on a barrier, and false otherwise. Its use will be described below.

The following sections expand this high level protocol idea by describing the details for some resources such as locks, semaphores and barriers are dealt with.

### 3.4.4 Incorporating Locks

Since the protocol layer intercepts all function calls between the application and the shared memory system, it is possible to replace the regular locking routines used by the application with special versions that follow the checkpointing protocol. These routines must provide the same synchronization behavior provided by the original routines and use the native implementation's locking routines to do so. The original routines are assumed to be called `native_acquire_lock`, `native_nonblock_acquire_lock` and `native_release_lock`. `native_nonblock_acquire_lock` is the non-blocking version of `native_acquire_lock`, which returns in a finite time and reports whether it has acquired the lock. They are replaced with `protocol_acquire_lock`, `protocol_release_lock` and `protocol_nonblock_acquire_lock`, respectively. The pseudocode for the protocol's modified lock acquire routines and the release and reacquire routines are shown below.

```
void forcerelease_lock(lock l) {
    omp_unset_lock(l);
}
```

```
void reacquire_lock(lock l) {
    omp_set_lock(l);
}

void protocol_acquire_lock(lock l) {
    native_acquire_lock(l);
    while (checkpoint_initiated!=0) {
        native_release_lock(l);
        perform_checkpoint(FALSE);
        native_acquire_lock(l);
    }
    record_resource_held(l);
}

int protocol_nonblock_acquire_lock(lock l) {
    int acq_success = native_nonblock_acquire_lock(l);
    while (checkpoint_initiated!=0) {
        native_release_lock(l);
        perform_checkpoint(FALSE);
        acq_success= native_nonblock_acquire_lock(l);
    }
    record_resource_held(l);
    return acq_success;
}
```

```

void protocol_release_lock(lock l) {
    remove_resource_held(l);
    native_release_lock(l);
}

```

The routines for forcefully releasing and reacquiring locks during checkpointing are trivial; they simply call the corresponding native routines.

Similarly, the routine `protocol_acquire_lock` calls `native_acquire_lock` in order to acquire the lock. Once the lock has been acquired, the thread checks to see if the checkpointing protocol has been initiated by checking if `checkpoint_initiated` is  $\neq 0$ . If so, it is assumed that the thread has been awoken so that it can take a checkpoint. In this case, it releases the lock that it was given in order to wake it up, performs a checkpoint and again attempts to acquire the lock. If the thread acquires the lock without the checkpointing protocol being initiated, then the `checkpoint_initiated` flag is read as 0 and the lock acquisition is complete and the lock ownership event is recorded by the protocol.

`protocol_nonblock_acquire_lock` is implemented very similarly to `protocol_acquire_lock` even though it is not a blocking operation. However, it is possible for the application to block on a lock by using repeated calls to `protocol_nonblock_acquire_lock`. As such, it is necessary for calls to `protocol_nonblock_acquire_lock` to regularly check the value of `checkpoint_initiated` to make sure that a checkpoint has not yet begun. Since `protocol_nonblock_acquire_lock` calls themselves are non-blocking, it is equally correct to check the value of `checkpoint_initiated` once every  $c$  calls to `protocol_nonblock_acquire_lock`, where  $c$  is some constant.  $c = 1$  in the above pseudo-code.

To release a lock the protocol records the fact that the thread no longer owns the lock and calls `native_release_lock` to release the lock itself.

Other similar types of resources such as critical sections can be dealt with in a very similar way.

```
while(1) {
    protocol_acquire_lock(1);
    ... code ...
    protocol_release_lock(1);
}
```

Figure 3.6: Locks example

For a more intuitive understanding of the protocol as it applies to locks, consider the code in Figure 3.6 being executed by two threads. Suppose that at some point in time Thread 0 acquires lock  $l$  and decides to take a checkpoint (i.e. calls `perform_checkpoint(FALSE)`) while still holding the lock. Meanwhile, thread 1 calls `protocol_acquire_lock` and blocks on lock  $l$ . In order to wake thread 1 up, thread 0 increments `checkpoint_initiated` and releases the lock. This causes thread 1 to acquire  $l$ , wake up and check the `checkpoint_initiated` counter. Since the counter is  $\neq 0$ , thread 1 knows that it was woken up in order to be checkpointed. As such, it lets go of lock  $l$  and takes a checkpoint. Meanwhile, thread 0 finishes its checkpoint, making sure to reacquire lock  $l$  before the second checkpoint barrier, and resumes its execution. After thread 1 has finished its checkpoint (and passed the second checkpoint barrier), it attempts to acquire the lock again.

Since thread 1 may not acquire lock  $l$  again before thread 0 decides to take another checkpoint, it may need to loop inside of `protocol_acquire_lock` multiple times before it can actually acquire the lock without being forced to checkpoint. Note that although thread 0 released lock  $l$  in the middle of the lock-protected region of code, application semantics were not violated because the protocol guarantees that by the end of each checkpoint resources are always restored to their pre-checkpoint owners. (i.e. since thread 0 held  $l$  when it called `perform_checkpoint`, it will hold  $l$  when `perform_checkpoint` returns)

### 3.4.5 Incorporating Semaphores

The protocol layer intercepts all calls to the semaphore wait and decrement functions and replaces them with its own implementation that enforces semaphore synchronization semantics and also implements the checkpointing protocol. The original semaphore functions are assumed to be called `native_wait_sem` and `native_decr_sem` while their replacements are called `protocol_wait_sem` and `protocol_decr_sem`, respectively. Semaphore objects of type `semaphore` are also wrapped by protocol semaphore objects `protocol_semaphore` that contain a reference to the semaphore and a `decr_sem` field that records whether the semaphore's counter has been forcibly decremented in the current checkpoint.

```
private lock sema_lock;

void forcerelease_sem(protocol_semaphore s) {
    // provide mutual exclusion for the decrement logic
    native_acquire_lock(sema_lock);

    // if the semaphore has not yet been forcibly decremented
```

```
        if(!s->decr_sem)
        {
            // forcibly decrement it and remember that this
            // semaphore has been decremented
            native_decr_sem(s->native_sem);
            s->decr_sem=TRUE;
        }
        native_release_lock(sema_lock);
    }

void reacquire_sem(protocol_semaphore s) {
    // provide mutual exclusion for the re-increment logic
    native_acquire_lock(sema_lock);
    // if the semaphore has not yet been re-incremented
    if(!s->decr_sem)
    {
        // re-increment it and remember that this semaphore
        // has been re-incremented
        native_wait_sem(s->native_sem);
        s->decr_sem=FALSE;
    }
    native_release_lock(sema_lock);
}

void protocol_wait_sem(protocol_semaphore s) {
```

```

native_wait_sem(s->native_sem);

while (checkpoint_initiated) {
    native_decr_sem(s->native_sem);
    perform_checkpoint(FALSE);
    native_wait_sem(s->native_sem);
}

}

void protocol_decr_sem(protocol_semaphore s) {
    native_decr_sem(s->native_sem);
}

```

`forcerelease_sem` is called by every thread in the application because there is no concept of semaphore "ownership" and any thread may have incremented or decremented the semaphore counter in the past. The primary job of `forcerelease_sem` is to decrement the counter at least once so that threads that may be blocked on the semaphore will wake up and realize that it is time to take a checkpoint. Since it will then immediately decrement the counter, any other threads that may be blocked on the semaphore are guaranteed to eventually wake up and take a checkpoint. The protocol above decrements the counter exactly once by recording in each semaphore's `->decr_sem` field whether the semaphore has been decremented during the current checkpoint and protecting this decrement operation with a special lock. Before the end of the checkpoint semaphores are re-incremented using the reverse procedure, guaranteeing that each semaphore is re-incremented exactly once and that each semaphore's `decr_sem` field is reset to `FALSE` at the end of each checkpoint.

The remaining protocol logic contained in functions `protocol_wait_sem` and `protocol_decr_sem` work exactly the same as in the locks case in Section 3.4.4 above.

### 3.4.6 Incorporating Barriers

The protocol layer intercepts application barrier calls and replaces them with calls to its own implementation of barrier. These are assumed to be global barriers (i.e. each barrier must be called by all threads) but variants of this protocol can be applied to different types of barriers. This implementation maintains all the synchronization semantics of the native barrier call (calling the native barrier internally to do so) but also internally implements the checkpointing protocol. The original barrier routine is assumed to be called `native_barrier` while the protocol's replacement routine is called `protocol_barrier`. The pseudocode for the barrier routine and the barrier release routine are shown below.

```
shared int threads_awoken[2] = { FALSE, FALSE };
```

```
private int barrier_id = 0;
```

```
void forcerelease_barrier() {
    threads_awoken[barrier_id] = TRUE;
    native_barrier();
}
```

```
void reset_barrier()
{
    threads_awoken[barrier_id] = FALSE;
    barrier_id = !barrier_id;
```

```

}

void protocol_barrier() {
    native_barrier();
    while (threads_awoken[barrier_id])
    {
        perform_checkpoint(TRUE);
        threads_awoken[barrier_id] = FALSE;
        barrier_id = !barrier_id;
        native_barrier();
    }
    barrier_id = !barrier_id;
}

```

The flags `threads_awoken[]` are used to determine whether or not a barrier completed because all threads arrived at the barrier (`=FALSE`) or because a thread called `forcerelease_barrier` to awaken any thread that was blocked at the barrier (`=TRUE`). `barrier_id` is a private variable that is used to choose the `threads_awoken` flag that is checked at each given point in order to prevent race conditions between adjacent checkpoints. It alternates between 0 and 1 every time each thread passes a checkpoint or a barrier, making sure that two threads may not write different values to the same `threads_awoken` flag unless the writes are separated by barriers.

The purpose of the `at_barrier` parameter to `perform_checkpoint` can now be explained. When a thread is released from a barrier to take a checkpoint, it does not need to call `forcerelease_barrier` since the fact that it was released in this

way implies that it and all the other threads that were blocked on this global barrier have been awoken. Thus, all threads previously blocked on barrier calls are now awake and ready to take a checkpoint. As such, in the call to `perform_checkpoint`, calls to `forcerelease_barrier` and `reset_barrier` are skipped since `at_barrier` is true.

### 3.4.7 Implicit Synchronization

It is possible for an application to implement its own form of synchronization. (spinlocks, Decker’s Algorithm, etc. [190]) Our checkpointing framework may deadlock on applications that use such implicit synchronization. The reason is fundamental - the protocol assumes that there is a mechanism for releasing resources and barriers. Since the protocol does not monitor or intercept the application’s individual reads and writes, it has no way to release threads that are blocked at application-implemented synchronization. Indeed, this is a property of any general checkpoint protocol since such a protocol may be implemented in software (since it must work above software shared memory) and there is no efficient way for a software-implemented protocols to track every relevant shared read and write operation. As such, the Restrict methodology is used to prohibit the application from using its own form of synchronization.

### 3.4.8 Eager vs. Lazy

When a thread acquires a lock in `protocol_acquire_lock`, it will take a checkpoint if the `checkpoint_initiated` counter  $\neq 0$ . This is true whether the thread was awoken by another thread thread calling `forcerelease_lock` or not. We call this an “eager” checkpointing mechanism. However, from the perspective of cor-

rectness, it is not imperative for a thread to take a checkpoint at lock acquisition time unless that lock was specifically released as part of a checkpoint. As such, an alternative would be to replace a global `checkpoint_initiated` with a number of flags, one for each lock (`lock.checkpoint_initiated=true` if its owner has taken a checkpoint and false otherwise). A given lock's `checkpoint_initiated` flag could be set inside of `forcerelease_lock` and a thread would only forcibly checkpoint if it tried to acquire a lock whose `checkpoint_initiated` flag was already set. We call this a “lazy” checkpointing mechanism (the same classification applies to other similar resources like critical sections and semaphores).

There are performance trade-offs that can be made between these two approaches. For some applications, an eager approach may give better performance because threads spend less time waiting for other threads to checkpoint. For other applications, a lazy approach may give better performance because threads are more likely to take checkpoints only the most optimal times/locations. For still other applications, a hybrid approach may give the best performance.

The rest of this chapter, including the experimental results focuses on the purely lazy variant of this protocol.

### 3.5 Applying the Protocol to OpenMP

The protocol described above is both simple and applies to a broad variety of shared memory APIs, memory models and implementations. However, without experimental evidence that it can be applied to real shared memory APIs in practice, the above is merely an empty statement. For one thing, it is not clear that it is possible to save and restore the state of the application running on top of arbitrary shared memory APIs. Another potential problem is that while the generic protocol

is applied in the above section to a number of common synchronization constructs, arbitrary APIs may implement very complex synchronization constructs that cannot be supported by this protocol.

The remainder of this chapter presents such an experimental validation, showing how the basic protocol above has been adapted to OpenMP, a popular shared memory API that is frequently used in scientific computing. In addition to its popularity and broad availability on a wide variety of platforms, OpenMP presents a number of complex features not present in the basic shared memory model discussed thus far. One major feature is parallel loops, where multiple threads may execute the iterations of a given for loop. Another new feature are **ordered** regions, which are a synchronization construct that works like critical sections, except that **ordered** regions inside a given parallel for loop must be executed in-order relative to the iterations of the loop. While the adaptation described here provides full support for all mandatory features of OpenMP, it only provides trivial support for nested parallelism, providing only a single thread to each nested parallel region.

The adapted protocol was implemented as part of the Cornell Checkpointing Compiler ( $C^3$ ) [53] system for application-level checkpointing. This enhancement enabled  $C^3$  to checkpoint real OpenMP applications and again show the practicality of this approach by showing that checkpointing could be provided for a variety of OpenMP implementations on a variety of software/hardware platforms, with no knowledge of these platforms' underlying details.

By providing concrete evidence that the above protocol is in fact generic and adaptable to real-world shared memory APIs, we hope to encourage the shared memory community to further pursue generic techniques for shared memory roll-back restart with the ultimate goal of making this capability available to arbi-

trary applications running on top of all types of shared memory. This section outlines the approach taken to adapting the protocol to OpenMP. Section 3.6 discusses the issues involved in saving the state of OpenMP applications, including internal OpenMP library state that is made visible to the application via the OpenMP API. Section 3.7 extends the protocol to OpenMP synchronization constructs that were not addressed in the discussion above. Section 3.8 provides a fully detailed discussion of the techniques in Sections 3.6 and 3.7. Finally, Section 3.9 provides the results of experiments using  $C^3$  with the SPLASH-2, NAS and EPCC benchmark suites, running on a 2-way Linux/Athlon platform and 4-way Linux/IA64, Tru64/Alpha and Solatis/Sparc platforms. These experiments show that  $C^3$  adapted to OpenMP shows low overheads for most applications it was tested with and identifies a few specific overheads that can be further improved. Section 3.10 summarizes the lessons learned from extending the above protocol to OpenMP.

### 3.5.1 Architecture

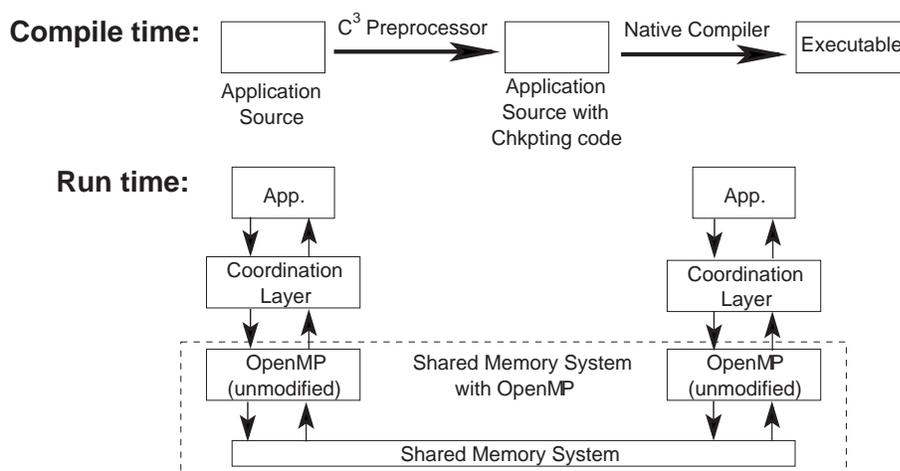


Figure 3.7: Overview of the  $C^3$  OpenMP Checkpointer

Figure 3.7 describes the  $C^3$  approach. The OpenMP API is defined as a set of

compiler `#pragma` directives and several functions that describe how a given program should be parallelized. As such, the only way to insert the protocol layer between the application and the OpenMP implementation is to perform compiler transformations on the source code that convert the original OpenMP directives into modified versions. The  $C^3$  pre-compiler reads the source files of OpenMP application written in C and instruments them to execute the checkpoint coordination protocol and to perform application-level saving of shared and private state. The  $C^3$  runtime is then placed as a layer between this modified executable and the native OpenMP library, performing additional checkpoint coordination and state saving functionality. This structure makes checkpointing a property of the application rather than of the underlying system, making it available on any platform on which the application is executed. The only modification programmers make to source files is to insert calls to function `ccc_potential_checkpoint` at program points where checkpoints may be taken. These points should ideally have minimal live state and need to be executed at least as often as the desired checkpoint interval. In practice placing potential checkpoint locations at the top of the application's main computation loop usually satisfies these conditions. Note that checkpoints need not be taken at every `ccc_potential_checkpoint` call; instead, the choice can be based on a timer or an adaptive mechanism such as [158] and [133].

The output of the pre-compiler is compiled with the native compiler on the hardware platform, and linked with a library that implements the above protocol for generating consistent snapshots of the state of the computation. This layer sits between the application and the OpenMP runtime layer, and intercepts all calls from the instrumented application program to the OpenMP library. This design

makes  $C^3$  compatible with any implementation of OpenMP and makes it possible to combine this shared-memory checkpointer with the MPI checkpointer described in Section 2.5 to provide checkpointing for hybrid MPI/OpenMP applications.

The  $C^3$  system implements the basic protocol in Figure 3.5 by supporting all synchronization constructs available in OpenMP and providing functionality to save the private and shared state of the application at the application-level, without saving the state of the native OpenMP implementation.

### 3.5.2 State classification

To apply this basic approach to the general class of OpenMP programs, it is necessary to describe what is done in the **Save State** step in Figure 3.3. Running OpenMP processes have four types of state:

- **Application state:** Includes application variables (both global and local), heap objects and the stack activation frames that exist at checkpoint-time.
- **Hidden state:** Any state inside the OpenMP runtime system that must be recreated on restart. Includes the locations of privatized and reduction variables, the binding between stack regions and thread numbers, and the scheduling of worksharing constructs.
- **Synchronization state:** The state of any synchronization objects that any threads held or was blocked on at checkpoint-time. These include barriers, locks, critical sections, and ordered regions in worksharing constructs.

### 3.5.3 The 4R's Applied to OpenMP

The 4R's methodology described in Section 1.1.2 considers the various techniques a checkpointing solution may use to save and restore the state of the application given the degrees of access it has to various portions of application state. This methodology is very relevant to the problem of checkpointing the state of OpenMP applications at the application-level (i.e. by checkpointers that transform the application to checkpoint its own state) because such a checkpointer has very different types of access to different portions of such applications' state. For example, the application has complete access to its process counter. It knows its value at all times and can use loops or gotos to modify the process counter at will. The same is not true for heap objects or the ID of each thread. While the application is completely aware of where its heap objects are located and the IDs of its threads, it has no power to choose where `malloc` should place an object or what ID to give to a particular thread. Finally, some types of state like the location of reduction variables are completely hidden from the application until the reduction is complete.

More specifically, consider the different state saving/restoring techniques and how they apply to different types of OpenMP application state:

- **Restore:** State that can be directly manipulated by the application and can therefore be directly saved at checkpoint-time and restored on restart. Example: Global and local variables.
- **Replay:** State that cannot be directly manipulated by the application but can be recreated using a sequence of deterministic operations. On restart it can be regenerated by replaying these operations. Example: OpenMP locks since their state is inaccessible to the application (the implementation details

of the `omp_lock_t` structures are unknown) but can be recreated using lock routines such as `omp_init_lock` and `omp_set_lock`.

- **Reimplement:** State that cannot be recreated by replaying operations. In this case the operations that create and maintain this state need to be reimplemented so that this state can be saved and restored. Example: heap objects in C since on restart it is not possible to force `malloc` to place these objects at their original locations. We have developed our own self-checkpointing implementation of the heap functions.
- **Restrict:** State that cannot be recreated by reimplementing the operations. This type of state cannot be supported and the application must be restricted from using this kind of state or to only using it in a restricted manner. Example: the use of application-implemented synchronization inside the application can cause our checkpointing protocol to deadlock and thus, its use must be restricted.

In the design of the  $C^3$  OpenMP checkpointer we have focused on achieving high performance via a strategy that uses the least invasive type of checkpointing mechanism possible. (i.e. Replay before Reimplement, Restore before Replay, etc.) While this has the downside of a more complex solution, our experiments (Section 3.9) validate this approach.

### 3.5.4 Discussion

At a high-level,  $C^3$  uses Restore to recreate the value of variables and heap objects, Replay to recreate activation frames, and Reimplement to recreate the heap. The approach for recreating the stack is discussed in Section 3.6.1.3. The remaining

categories of Hidden state and Synchronization state are described in detail in Sections 3.6 and 3.7, respectively. Not described here are the details of how  $C^3$  handles the remaining constructs of OpenMP (e.g., master regions, flush directives, and the time functions) as these details are trivial.

## 3.6 Hidden State

### 3.6.1 Threads

The parallel directive creates new threads and assigns them memory regions for their stacks. Each thread can determine its own thread number and the number of threads currently running. During a checkpoint enough information must be saved so that on restart the threads can be recreated and the OpenMP runtime system can be reset into a state that, from the application's point of view, is equivalent to its state at the time of the checkpoint.

#### 3.6.1.1 Recreate threads and their IDs

On restart the application must recreate the same number of threads as were present at checkpoint-time. In OpenMP new threads are created when some parent thread passes through a `#pragma omp parallel` directive. To recreate the same threads on restart we have the application pass through the `parallel` directives that had created the threads it had at checkpoint-time (`omp_set_num_threads` is used to recreate the same number of threads). Thread IDs are generated deterministically.

### 3.6.1.2 Recreate thread to stack mapping

The application may have pointers to stack objects. Since  $C^3$  does not use a mechanism for retargeting these pointers on restart<sup>2</sup>, they will become invalid if the objects move. As such, it is necessary that each thread be reassigned the same stack region that it had in the original execution. Although `parallel` directives can recreate threads, OpenMP provides no guarantees about the locations of their stacks, nor lets these be set explicitly. However, in practice OpenMP implementations operate under some reasonable assumptions.

Suppose that we assume that when a `parallel` construct is re-executed on restart that the set of stacks allocated to the threads will be the same as during the original execution (even if the thread to stack mapping is different). In this case, thread 0 may be assigned the stack that was assigned to thread 1 in the original run. By providing a custom implementation of `omp_get_thread_num`, we can shield the application from this change by providing a mapping of stack regions to thread ids that is that same as during the original execution.

Let us relax this assumption to allow the stacks on restart to move relative to the original stacks by a small amount (less than a few KB). In this case it is possible to pad the start of each stack with a buffer and on restart use `alloca` to pad it with the right number of bytes to make the starting point of the new stack line up to the old stack from the application's point of view.

If neither assumption holds, we can reimplement the thread to stack assignment mechanism. In many versions of UNIX and Linux, this can be done using the `mmap` and `setcontext` system calls. Windows has equivalent system calls. More details about these mechanisms are provided in Section 3.8.2.6.

---

<sup>2</sup>For portable checkpointing, such a mechanism is used. See [83] for details.

To summarize, depending upon the assumptions that can be made about how a particular implementation assigns stacks to threads, it is possible to use varying degrees of Replay and Reimplementation to ensure that threads are assigned the same stacks on restart. The most general and portable approach requires a complete reimplementation of this feature.

### 3.6.1.3 Recreate the stack contents

After we have recreated the threads and re-associated each thread with its correct stack and thread ID, the contents of each stack must be recreated. This consists of:

- Recreating the activation frames that were present on the stack when the checkpoint was taken,
- Ensuring that stack allocated variables are placed at the same locations, and
- Restoring the values of stack allocated variables.

We describe each in more detail below.

**Recreating the activation frames** Since the developer may place checkpoint locations at arbitrary points in the source code, checkpoints can occur within nested function calls, including regions encapsulated by OpenMP directives (e.g. within parallel regions). While the application can restore the stack regions that it created, it cannot restore the stack regions created by entering OpenMP regions since their size and contents are determined by the OpenMP implementation and their presence may correspond to additional state inside of OpenMP. As such, the stack is recreated via Replay by calling the function calls and reentering the

OpenMP directives that were on the stack at checkpoint-time. This is done by adding statement labels to function calls and OpenMP directives that can reach a checkpoint location and using these labels on restart to invoke the same functions whose frames were present when the checkpoint was taken, as described in detail in Section 3.8.2.5.

**Ensuring the location of stack allocated variables** To preserve the validity of application pointers, on restart all stack allocated variables must be assigned to their original addresses. Section 3.6.1.2 discusses how threads are assigned their original stack regions. Thus, all that remains is to ensure that all the function activation frames and OpenMP directives are placed at the addresses they were at checkpoint-time.

This is trivial for function calls since replaying their calls on restart will result in activation frames that are exactly the same size as they were at checkpoint time. It is more complex for OpenMP directives since there is no guarantee that when a given OpenMP directive is reentered on restart it will take up the same amount of stack space as during the original execution. The solution is to use `alloca` to pad the stack during the original execution, using the mechanism from Section 3.6.1.2 above to align restart stack starting addresses to their original values. Section 3.8.2.6 provides more details about this mechanism.

**Restoring stack allocated variables** Once the stack-allocated application variables have been placed at their original memory locations, it is possible for the application to directly restore their values from the checkpoint. The details of how this is done are provided in Section 3.8.2.7.

### 3.6.2 Privatized and Reduction Variables

In OpenMP there exist variables whose locations cannot be deterministically assigned. These are the explicitly privatized variables and the reduction variables. The reason for this is that the specification does not explicitly say how these variables are allocated and assigned addresses. Because of this, Replay is not an option and our system relies on Reimplementation, which is performed by the precompiler. The details of the transformation can be found in Section 3.8.2.4; the net effect is that every program variable is assigned a location in the globals, stack or heap, to which it can be deterministically reassigned on restart.

### 3.6.3 Worksharing constructs

The final category of OpenMP constructs whose state is hidden to the application are the *worksharing* constructs, which include `single`, `sections`, and `for` constructs. These constructs assign work units to threads using various scheduling choices. The hidden state is this assignment and the completion status of each work unit.

In the most general sense, these constructs cannot be handled using a Replay mechanism. The reason is that the assignment of work units to threads is inherently non-deterministic. Consider the example shown in Figure 3.8. In this example, only one thread will be assigned to execute the body of the `single` construct. Suppose that in the original run of the application, this `single` construct is assigned to thread  $t$ . On restart, there is no way to re-execute the `single` construct and ensure that it would again be assigned to thread  $t$ .

Because Replay is not possible, one possibility would be to Reimplement the worksharing constructs. In this case, checkpointing the reimplemented worksharing

```

#pragma parallel {
    #pragma omp single
    {
        ...
        perform_checkpoint();
        ...
    }
}

```

Figure 3.8: Single Construct Example

constructs would become simple: our implementation would directly save and restore the scheduling data structures.

However, rather than fully Reimplement worksharing constructs (which may result in reduced performance), it is possible to deal with them via a mixture of Replay and Reimplement. Consider the example shown in Figure 3.9. Suppose that every thread takes its checkpoint within this construct. On restart, we must (i) ensure that each thread will resume executing the same work unit within which it took its checkpoint and (ii) make sure that the remaining unexecuted work units will be scheduled to run on some threads. The restart code is shown in Figure 3.10. To deal with the first issue  $C^3$  extracts the loop's body and directly jumps to the location inside where the checkpoint was taken (Reimplement). Once these work units have been completed, the native OpenMP `for` construct is used to schedule unexecuted work units (Replay). Since OpenMP requires that the loop inside a `for` construct have a very simple form, the remaining sparse iteration space is mapped into a dense form (into array `remaining_iters[]`). Note that since this solution does not depend on which work units were executed, in-progress or remaining, it

works with all loop scheduling policies. This approach is illustrated in Figure 3.10

```
#pragma omp for
for(i=0; i<1000; i++)
{ ... loop body ... }
```

Figure 3.9: For Construct Example

```
i = iter_at_checkpoint(thread_id);
{ ... loop body ... }
// wait for the other threads to finish.
ccc_global_barrier();
remaining_iters[*] =
    condense_remaining_iters();
#pragma omp for
for (ii=0; ii<num_remaining_iters; ii++) {
    i = remaining_iters[ii];
    { ... loop body ... }
}
```

Figure 3.10: Restarting a simple for construct

The above mechanism works for worksharing constructs without a `nowait` clause (i.e. terminated by an implicit barrier). However, in situations where the `nowait` clause is present it is possible for different threads to checkpoint inside of different worksharing constructs. For example, in the code shown in Figure 3.11, it is possible for some threads to checkpoint inside the first `for` loop while others checkpoint inside the second. Thus, on restart some threads will enter both work-

sharing regions while others will enter only the second. This is illegal in OpenMP, which requires all threads to execute the same sequence of worksharing constructs.

This restriction makes it impossible to use native OpenMP `for` loops for scheduling the unfinished iterations of the first loop, as was done in the base solution of Figure 3.10. Instead, the base restart algorithm is extended as follows.

Let  $W$  be the latest worksharing construct that any thread took a checkpoint inside of (worksharing constructs are counted from the start of the application as threads enter them; latest=construct with largest count). For all worksharing constructs that precede  $W$ , their unexecuted work units are scheduled using our own implementation (Reimplement).  $W$ 's unexecuted work units are scheduled using the mechanism in Figure 3.10 (Replay).

```
#pragma omp for nowait
for(i=0; i<1000; i++)
{ ... ccc_checkpoint(); ... }

#pragma omp for nowait
for(i=0; i<1000; i++)
{ ... ccc_checkpoint(); ... }
```

Figure 3.11: Multiple For Construct Example

Full details for how worksharing constructs are checkpointed are provided in Section 3.8.4.

## 3.7 Synchronization State

While the generic protocol described in Section 3.4 covers a several different types of synchronization constructs, it does not address a number of  $C^3$ - and OpenMP-specific issues. In this section it is extended into a full checkpointing protocol for OpenMP.

### 3.7.1 Potential Checkpoint Locations

The basic protocol said little about when checkpoints could be initiated, except to say that they could not be initiated by threads while they were in the middle of calling shared memory API functions.  $C^3$  is an application-level checkpointer, meaning that it uses potential checkpoint locations that have been explicitly identified in the application source code as the only possible places a checkpoint may be taken.

Thus, the pseudocode in Figure 3.5 must be augmented with the pseudocode in Figure 3.12 below. Function `potential_checkpoint` is called every time an application thread reaches a potential checkpoint location. `time_to_checkpoint()` decides whether it is time to take a checkpoint by using a timer, user guidance, an adaptive mechanism [158] [133] or some other technique.

```
void potential_checkpoint() {
    if(time_to_checkpoint())
        perform_checkpoint(FALSE);
}
```

Figure 3.12: Checkpointing Protocol Extended with Potential Checkpoint Locations

Although OpenMP provides a barrier directive, it is not used by the  $C^3$  run-

time or the transformed application. This is because the OpenMP specification precludes the use of the barrier directive in certain constructs (e.g., work-sharing, critical, ordered or master). Since developers may place checkpoints within these constructs,  $C^3$  uses its own implementation of barriers in the routine `Global_Barrier` and in the transformed application to avoid such violations. This is explained in more detail in Section 3.8.2.3.

### 3.7.2 Incorporating OpenMP Synchronization Constructs

OpenMP provides applications with several synchronization constructs: `locks`, `barriers`, `atomic` updates, `critical` regions and `ordered` regions. It also allows for very limited support for application-implemented synchronization.

Sub-protocols for locks and barriers have already been discussed in Sections 3.4.4 and 3.4.6. These sub-protocols can be applied directly to OpenMP locks and barriers, with `native_acquire_lock` corresponding to `omp_set_lock`, `native_nonblock_acquire_lock` corresponding to `omp_test_lock`, `native_release_lock` corresponding to `omp_unset_lock` and `native_barrier` corresponding to `#pragma omp barrier`.

#### 3.7.2.1 Incorporating Atomic Updates

OpenMP `atomic` updates are expressions of the following format:

```
#pragma omp atomic
    var+=func();
```

While OpenMP provides no synchronization guarantees about the call to `func`, it does provide mutual exclusion for the `+=` operation. Both the read and the write

are performed inside a critical section that allows no other atomic update of *var* to happen at the same time. Thus, it is possible for threads to simultaneously perform multiple `atomic` updates with no fear of data races.

Since  $C^3$  allows for checkpoint locations to be placed anywhere in the application, it is possible for them to appear inside of `func`. However, since the `atomic` directive provides no synchronization for the execution of `func`, it is legal to hoist it out of OpenMP atomic directives, legally ensuring that potential checkpoint locations never occur in `atomic` directives. This transformation is expected to have no overhead because the semantics of the two versions of code are identical.

### 3.7.2.2 Incorporating Critical Regions

Since `critical` and `ordered` regions are conceptually similar to locks, they are incorporated into the framework in a similar fashion (Section 3.4.4). There are several key differences.

First, OpenMP has no routines for explicitly acquiring or releasing `critical` or `ordered` regions. Instead, they are blocks of code tagged by a `#pragma` that identifies them as `critical` or `ordered`. As such, entering and exiting a `critical` or `ordered` region is done implicitly by entering and exiting the corresponding block of code. This means that in order to release and reacquire these resources the routines `forcerelease_critical/forcerelease_ordered` and `reacquire_critical/reacquire_ordered` must execute code to exit and enter these code blocks.

For example, in the program in Figure 3.13, suppose a checkpoint was taken inside the `critical` section in function `bar` after `bar` was called from `foo`. Since the the checkpointing thread will be holding both `critical` regions A and B, it will

need to first release them and later on reacquire them. This is done by first jumping from the checkpoint location to the end of `critical` region B, exiting B, jumping to the end of `critical` region A, exiting A and finally taking a checkpoint. In reacquiring the `critical` regions the same things are done in reverse: the thread reenters `critical` region A, jumps to the start of `critical` region B, reenters B, jumps to the original checkpoint location and finishes the checkpoint.

```
bar() {  
    #pragma omp critical(Name_B)  
    {  
        ...  
        potential_checkpoint();  
        ...  
    }  
}  
  
foo() {  
    #pragma omp critical(Name_A)  
    {  
        ...  
        bar();  
        ...  
    }  
}
```

Figure 3.13: Critical example

This can be done by using a set of control flags and introducing additional `return` and `goto` instructions into the application source code. `setjmp` and `longjmp` may be used as well. Section 3.8.3.8 provides the details how this is done for `critical` regions.

The second key difference is that when a `critical` or `ordered` section is re-executed either during restore or replay, it must be ensured that the local variables in the enclosed scope are given the same addresses that they had during the original execution. The OpenMP specification does not guarantee this, but it can be handled using `alloca` in a manner similar to the approach in Section 3.8.2.6 for ensuring that stack regions are correctly reassigned on restart.

The case of `ordered` regions is complicated by their special semantics: if an `ordered` region appears inside a parallel `for`, its code must be executed in iteration order (i.e. if  $i < j$  then the `ordered` region in iteration  $i$  must be executed before the `ordered` region in iteration  $j$ ). Suppose that thread  $t_1$  has decided to take a checkpoint inside an `ordered` region in iteration  $i$  while another thread  $t_2$  is blocked on entry into an `ordered` region in a later iteration  $j$  (where  $i < j$ ). In order to wake up  $t_2$  and force it to take a checkpoint, (i)  $t_1$  must exit its `ordered` region and (ii) we must make sure that all iterations upto  $j$  are passed so that  $t_2$  can wake up, enter its `ordered` region and realize that a checkpoint has begun. After the checkpoint has completed, the skipped iterations must be reallocated to some threads in a manner similar to how  $C^3$  restarts parallel `for` loops and executed normally. The details of this protocol are provided in Section 3.8.4.1.

### 3.7.2.3 Incorporating Application-Implemented Synchronization

OpenMP features a very loose memory model that provides very little functionality for implementing synchronization using anything other than OpenMP's dedicated synchronization operations like `#pragma omp barrier` and `omp_set_lock`. Although it provides a memory fence in the form of `#pragma omp flush`, OpenMP makes no guarantees of write atomicity (`#pragma omp atomic` is a specialized critical section, with no atomicity semantics), meaning that the only synchronizations that can reliably be implemented in OpenMP are variants of the following spinlock [52]:

Initially,  $flag = 0$

Thread 0	Thread 1
<pre>flag=1 #pragma omp flush</pre>	<pre>#pragma omp flush while(flag=0){     #pragma omp flush } print(flag)</pre>

The key thing about this spinlock is that thread 1 keeps looping until the value of `flag` changes. It does not matter what the value of `flag` is as long as it eventually changes to something that is not equal to 0. While the lack of write atomicity in OpenMP means that the value of `flag` is undefined for much of the above program's execution, the `#pragma omp flush`'s do guarantee that the value of `flag` read by thread 1 will eventually change. This is all that is required by the above spinlock example to work. However, if we were to add to the body of the while loop a statement that prints the value of `flag` during each iteration, this

would no longer produce sensible results since the printed values of `flag` could be literally anything (they are undefined by the OpenMP specification).

While this example shows that it is in fact possible to write functioning synchronizations in OpenMP, the very loose memory model makes such synchronizations mostly useless. Therefore, while the limitations of the checkpointing protocol force  $C^3$  to Restrict the use of such synchronization in the applications it may deal with, this restriction is expected to affect almost no real OpenMP applications.

## 3.8 Details of transformations and protocols

### 3.8.1 Overview

This section describes in detail the checkpointing protocol used by  $C^3$  as well as the source code transformations performed by  $C^3$ . The description is divided into four sections. The first part (Section 3.8.2) shows a series of transformations that are performed by the compiler. These are used to (i) record the state of the application, (ii) insert the  $C^3$  checkpointing layer between the application and the OpenMP library, and (iii) provide mechanisms for the checkpointing protocol to forcefully release and reacquire `critical` and `ordered` regions. Section 3.8.3 provides additional details about the checkpointing protocol itself, including more detailed pseudocode for `barriers` and `locks` (`critical` regions are not separately described since their major mechanisms are covered in other sections). Section 3.8.4 focuses on the details of checkpointing worksharing constructs, including the detailed protocol for `ordered` regions. Finally, Section 3.8.5 addresses some additional miscellaneous issues.

## 3.8.2 Transformations

### 3.8.2.1 Transformation #1: Directive body lifting

The first transformation lifts code that appears within OpenMP constructs into separate procedures. This is necessary to (i) allow us to reimplement OpenMP's privatization functionality (detailed in Section 3.8.2.4) and (ii) to make it possible to use `alloca` to correctly align portions of the call stack on restart (detailed in Section 3.8.2.6). When lifting code into separate procedures, it is necessary to ensure that local variables in scope are still accessible. That can be done using a transformation similar to *lambda lifting* [65], except that in our case closures do not have to be heap allocated.

The following code,

```
int glob;

void foo() {
    int fooLcl;
    ...
    #pragma omp parallel
    {
        int parLcl;
        ... parallel code
        ... glob ... fooLcl ... parLcl ...
        #pragma omp critical
        {
            int critLcl
```

```

        ... critical code ...
        ... glob ... fooLcl ... parLcl ... critLcl ...
    }
}
...
}

```

becomes

```

void parallel_body(int fooLcl);
void crit_body(int fooLcl, int parLcl);
int glob;

void foo()
{
    int fooLcl[20];
    ...
    #pragma omp parallel
    { parallel_body(fooLcl); }
    ...
}

void parallel_body(int* fooLcl)
{
    int parLcl;
    ... parallel code
    ... glob ... fooLcl ... parLcl ...
}

```

```

#pragma omp critical
{
    int parLcl_shadow;
    crit_body(fooLcl, parLcl, &parLcl_shadow);
    parLcl = parLcl_shadow;
}
}

void crit_body(int *fooLcl, int parLcl, int *parLcl_shadow) {
    int critLcl
    ... critical code ...
    ... glob ... *fooLcl ... *parLcl ... critLcl ...
    *parLcl_shadow = parLcl;
}

```

The above transformation passes all local variables available in the scope of the OpenMP directive as arguments into the function that contains the directive's body. These arguments can be passed in either by value or by reference. Arguments passed by value have additional costs in terms of memory (space on the stack) and time (cost of copying the variable). Such arguments need to have their values explicitly copied out from the copy inside the called function back to the original variable. Arguments passed by reference have minimal memory and time costs. However, passing and using these variables by references makes it difficult for the compiler to determine whether two such references alias each other. This can prevent it from performing certain optimizations, reducing the application's performance.

We have found that to balance these tradeoffs, it is sufficient to pass private scalar variables by value and private array variables by reference. All shared variables must be passed by reference. Since global variables are globally visible, they do not have to be explicitly passed into these functions and do not suffer from these performance tradeoffs.

### 3.8.2.2 Transformation #2: Parallel For Normal Form

Each `parallel for` directive is transformed into a `parallel` directive that contains a `for` directive. This is done because in many cases  $C^3$  transformations need to place additional code inside the `parallel` region but before the `for` loop. All clauses that appeared on the `parallel for` directive are moved to the resulting `parallel` and `for` directives, as appropriate.

Thus, the following code,

```
int glob;

void foo() {
    int red;
    ...
    #pragma omp parallel for private(glob) reduction(+: red)
    for(int iter=0; iter<10; iter++)
    {
        ... glob ... red ... iter ...
    }
    ...
}
```

becomes

```

int glob;

void foo() {
    int red;
    ...
    #pragma omp parallel private(glob)
    #pragma omp for reduction(+: red)
    for(int iter=0; iter<10; iter++)
    {
        ... glob ... red ... iter ...
    }
    ...
}

```

### 3.8.2.3 Transformation #3: Barrier Replacement

Because of OpenMP forbids the use of barriers inside of worksharing constructs, `critical`, `ordered` and `master` sections,  $C^3$  transforms the application source code to use  $C^3$ 's implementation of barriers. This includes replacing instances of `#pragma omp barrier` with calls to `protocol_barrier` and extracting all implied barriers and replacing those as well. Barriers are implied at the end of `parallel` regions and worksharing constructs without a `nowait` clause. Such regions are transformed to have a `nowait` clause and be followed by a call to `protocol_barrier`.

It is not possible to remove the implied barrier at the end of `parallel` regions since they do not have a `nowait` clause. As such, it is possible for the application

to deadlock if some thread has finished its parallel region and is blocked on this final barrier while some other thread is still executing and has decided to take a checkpoint. These deadlocks are avoided by placing a `protocol_barrier_end` call at the end of each parallel region. `protocol_barrier_end` is identical to `protocol_barrier` except that it only matches other `protocol_barrier_end` calls rather than regular `protocol_barrier` calls. When the `protocol_barrier_end` returns on any thread, it is known that all threads have finished the body of the parallel region and can enter its final implicit barrier.

Thus, the following code,

```
void foo() {  
    ...  
    #pragma omp parallel  
    {  
        #pragma omp for  
        for(int iter=0; iter<10; iter++)  
        {  
            ...  
        }  
        #pragma omp single nowait  
        {  
            ....  
        }  
        ...  
        #pragma barrier  
        ...  
    }  
}
```

```
    }  
}
```

becomes

```
void foo() {  
    ...  
    #pragma omp parallel  
    {  
        #pragma omp for nowait  
        for(int iter=0; iter<10; iter++)  
        {  
            ...  
        }  
        ...  
        protocol_barrier();  
        #pragma omp single nowait  
        {  
            ...  
        }  
        ...  
        protocol_barrier();  
        ...  
        protocol_barrier_end();  
    }  
}
```

### 3.8.2.4 Transformation #4: Privatization

This transformation reimplements OpenMP's explicit privatization functionality available through the `private`, `firstprivate`, `lastprivate`, `copyin` and `reduction` clauses, which may appear on OpenMP directives with bodies such as `parallel`, `for` and `single`. It also reimplements the implicit privatization done for `parallel for` loop iteration variables.

The purpose of the transformation is to allow for checkpointing of constructs where a user is allowed to explicitly or implicitly privatize variables that would otherwise be shared. OpenMP provides no semantics regarding where these new private copies will be allocated or whether the choice of address is deterministic. As such, it is not possible for an application-level solution to force these private variables to occupy the same locations on restart as they did during the original execution (i.e. Restore and Replay are not possible).

#### Local Variables

The `private`, `firstprivate`, `lastprivate`, and `reduction` clauses apply to variables in scope at their parent directive. The privatization of these variables is implemented by extending the Lamda Lifting technique from Section 3.8.2.1. Scalar variables are passed by value into the function that wraps the directive's body, creating private copies of each variable on each thread's stack. Array variables are passed in by reference and then copied into private local copies inside the body function.

Thus, the following code,

```
void foo()
{
    int fpv[20], pv, lpv, red, iter;
```

```

...
#pragma omp for firstprivate(fpv) private(pv),
                lastprivate(lpv), reduction(+: red)
for (iter=0; iter<10; iter++)
{
    ... fpv ... pv ... lpv ... red ... iter ...
}
....
}

```

becomes

```

void parallel_for_body(int* fpv_shared, int pv, int lpv,
                    int red, int iter);

void single_body(int cpv);

void foo()
{
    int fpv[20], pv, lpv, red, iter;
    ...
    #pragma omp for nowait
    for (int iter=0; iter<10; iter++)
    {
        parallel_for_body(fpv, pv, lpv, red, iter);
    }
    protocol_barrier();
    ...
}

```

```

}

void parallel_for_body(int* fpv_shared, int pv, int lpv,
                      int red, int iter)
{
    int fpv[20];
    memcpy(fpv, fpv_shared, sizeof(int)*20);
    ... fpv ... pv ... lpv ... red ... iter ...
}

```

### LastPrivate Variables

In order to enforce `lastprivate` semantics, the value of the private copy at the end of the sequentially last iteration of the associated `for` loop or the last section of the associated `sections` directive must be copied into the original shared variable. This is implemented by passing the original shared variable into the function by reference and performing the copy at the end of the final loop iteration. Thus, the transformation above is enhanced to produce the following output code (only the `lastprivate` variable is shown):

```

void parallel_for_body(int lpv, int* lpv_shared);

void foo()
{
    int lpv, iter;
    ...
    #pragma omp for nowait
    for (int iter=0; iter<10; iter++)

```

```

    {
        parallel_for_body(lpv, &lpv, iter);
    }
    protocol_barrier();
    ...
}

void parallel_for_body(int lpv, int* lpv_shared)
{
    ... lpv ... iter;
    ...
    if(iter==9) *lpv_shared = lpv;
}

```

### Reduction Variables

Reimplementing reduction functionality requires additional code at the end of each loop iteration to accumulate that iteration's value of the reduction variable into the main shared reduction variable. Since each thread has its own private reduction variable, that thread's contributions are accumulated into the variable. When the thread has finished with its iterations, its private reduction variable is accumulated into the shared reduction variable (this works because reduction operations are associative and commutative). The result of applying this transformation to the original example code is as follows (only the reduction variable is shown):

```

void parallel_for_body(int red, int *red_acc, int iter);

void foo() {

```

```
int red, iter;
...
{
    // each thread's private reduction accumulation
    // variable
    int red_acc=0;
    // initialize the shared reduction variable
    #pragma omp critical (C3Reduction)
        red=0;

    #pragma omp for nowait
    for (int iter=0; iter<10; iter++)
    {
        parallel_for_body(red, &red_acc, iter);
    }
    // accumulate each thread's contribution to the shared
    // reduction variable
    #pragma omp critical (C3Reduction)
        red+=red_acc;
}
protocol_barrier();
...
}

void parallel_for_body(int red, int *red_acc, int iter) {
```

```

    red = *red_acc;
    ... fpv ... pv ... lpv ... red ... cpv ... iter;
    // accumulate the new value into red_acc
    *red_acc = red;
}

```

### ThreadPrivate Variables

In OpenMP, global variables are shared by default. They can be privatized by using the `threadprivate` clause, which creates a private copy of a given global variable for each thread. When threads are created by the `parallel` directive, the values of their `threadprivate` variables are initially undefined. The `copyin` clause can be added to the `parallel` directive to initialize each thread's private copy to the value in the master thread's copy. These values are guaranteed to be preserved across different `parallel` regions as long as the number of threads is the same from one region to the next and cannot change dynamically.

$C^3$ 's reimplementations of `threadprivate` variables involves replacing the application's original `threadprivate` variables with pointers. Each thread's pointer is set to point to that thread's copy of the variable and all accesses to these variables are modified to use these pointers. Thread 0's private copy is a global variable (since it must exist outside of any `parallel` regions) while the private copies of all other threads are allocated on the heap using `malloc`. `copyin` is implemented by using `memcpy` to copy the data of the global variable to all the copies on the heap. These transformations are shown below.

The following code,

```

int tpv, tpv_cpin;

#pragma omp threadprivate(tpv, tpv_cpin);

```

```

void foo() {
    ... tpv ... tpv_cpin ...
    #pragma omp parallel copyin(tpv_cpin)
    {
        ... tpv ... tpv_cpin ...
    }
    ... tpv ... tpv_cpin ...
}

```

becomes

```

// records the maximum number of threads that have been
// generated in any prior parallel region
int maxThreadsSeen=1;

// the master thread's copies of the threadprivate variables
int ccc_tpv, ccc_tpv_cpin;

// initialize the variable to point to the master thread's
// copy
int *tpv = &ccc_tpv, *tpv_cpin = &ccc_tpv_cpin;

#pragma omp threadprivate (tpv, tpv_cpin, parSeen)

void foo() {
    ... *tpv ... *tpv_cpin ...;
}

```

```

#pragma omp parallel
{
    if(omp_get_thread_num()!=0)
    {
        // if this thread copy has not been created yet
        if(omp_get_thread_num()>=maxThreadsSeen)
        {
            // allocate private thread copies for the
            // threadprivate variables
            tpv = ccc_malloc(sizeof(int));
            tpv_cpin = ccc_malloc(sizeof(int));
        }
        // perform copyin functionality
        memcpy(tpv_cpin, ccc_tpv_cpin, sizeof(int));
    }

    ... *tpv ... *tpv_cpin ...;

    maxThreadsSeen=max(maxThreadsSeen,
                        omp_get_num_threads());
}
... *tpv ... *tpv_cpin ...;
}

```

### Comments

Care must be taken when performing this transformation to ensure that the all

references to transformed variables are still valid. In particular, certain OpenMP directives require the use of a variable name. If this transformation would replace references through this name with a reference through a pointer, then a temporary variable may have to be introduced in order to hold a copy of the reference through a pointer for the duration of the directive.

### 3.8.2.5 Transformation #5: Recreation of the stack and the threads

Since the application can maintain pointers to variables on the stack, it is critical on restart to restore the stack such that all application variables are placed at their original locations. This is made difficult by the fact that at checkpoint time, the stack can contain regions from both the application and the OpenMP implementation. Figure 3.14(a) provides a sample call stack of some thread. This thread started out in `main`, entered a `parallel` region, called a function `foo`, entered a critical section and finally called function `bar`. Each of the above uses up some amount of call stack space. Recreating this call stack configuration requires the application to use the Replay methodology to call the same functions and enter the same OpenMP regions as were present on the stack at checkpoint time.

To do this,  $C^3$  maintains a record of all relevant function calls and OpenMP regions currently on the stack. Each call and region entry point within a given function is assigned a unique id (unique within that function). To represent the current state of the stack,  $C^3$  records the ids of all function calls and region entry points corresponds to the function and regions currently on the stack. These ids are stored on the `pc_stack`, which is manipulated using functions `ccc_push_pc` and `ccc_pop_pc`.

To keep the `pc_stack` upto date with the state of the call stack,  $C^3$  surrounds

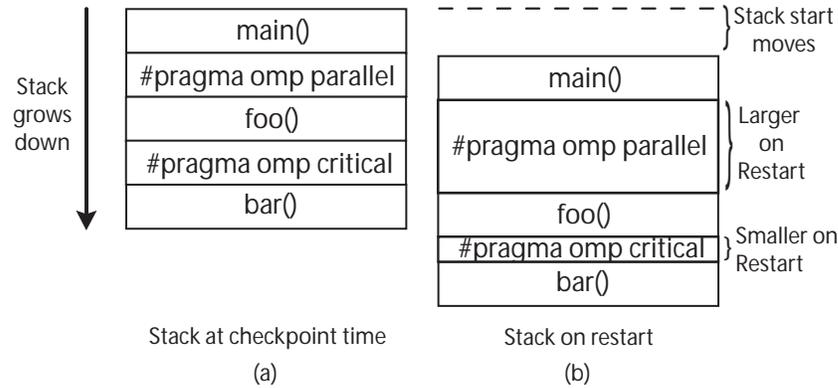


Figure 3.14: Sample application call stack

each function call and OpenMP region with calls to `ccc_push_pc` and `ccc_pop_pc`. `ccc_push_pc` is placed before function calls and region entry points and pushes onto the `pc_stack` the unique id of that location. `ccc_pop_pc` is placed immediately after function calls and region exit points to remove their record after they have been removed from the call stack. Thus, at all times the `pc_stack` contains the current configuration of the call stack and each thread's `pc_stack` is saved as part of the thread's checkpoint.

To recreate the call stack on restart,  $C^3$  calls and enters the same functions and OpenMP regions as were present on the call stack at checkpoint time. This is done by inserting before each function call and OpenMP region entry point, a label named `ccc_label_id#` where `id#` is the unique id of that call or entry point. On restart, when an application thread reenters some function that was on its call stack at checkpoint time (it always starts in `main`, which is always at the start of the call stack), it reads the next call/entry point id from the `pc_stack` and jumps to that call/entry point. Once the that function/OpenMP region is reentered, the same procedure is repeated until the application recreates its entire call stack.

Since maintaining the `pc_stack` has a runtime cost, one optimization used by  $C^3$  is to only maintain the `pc_stack` upto date for function calls/OpenMP regions

that may be on the stack at the time of a checkpoint. A simple call graph analysis can determine whether there exists a call path from a given function/OpenMP region to some potential checkpoint location or some OpenMP blocking synchronization call (since they may be forced to take checkpoints inside themselves). Thus, the above insertion of labels and calls to `ccc_push_pc` and `ccc_pop_pc` is performed only around function calls and OpenMP regions that may contain a checkpoint. Another optimization is to omit the calls to `ccc_push_pc` and `ccc_pop_pc` in functions that contain only one function call/OpenMP region that can contain a checkpoint since in those cases there is only one possible place to jump to.

Note that if a checkpoint was taken inside a `parallel` region, the entry into this region will be on the `pc_stack` and as such the master thread will reenter this region. This will create new threads, each of which will load their own `pc_stacks` and restore their own call stacks. Full details about thread recreation are presented in Section 3.8.3.2.

The following code,

```
void foo() {
    ...
    bar();
    ...
    #pragma omp baz
    {
        ...
    }
    ...
    #pragma omp parallel
```

```

    {
        ...
        protocol_barrier();
        ...
    }
    ...
}

void fib() {
    ...
    bat();
    ...
}

```

where the function ids “bar” and “bat” are any of the following

- `potential_checkpoint`
- `protocol_barrier`
- `ccc_set_lock`, `ccc_test_lock`
- Any function call that lead to a function call/OpenMP region of any functions/OpenMP regions listed here.

and the id “baz” is any of the following

- An OpenMP `critical` region
- An OpenMP `ordered` region

- Any OpenMP region call that lead to a function call/OpenMP region of any functions/OpenMP regions listed here.

becomes

```

void parallel_body();

// Function with multiple jump targets
void foo() {
    if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY) {
        // This fragment of code is called a "jump table".
        switch (*pc_stack_cursor++) {
            case 1: goto ccc_label_1;
            case 2: goto ccc_label_2;
            case 3: goto ccc_label_3;
        }
    }
    ...
    ccc_push_pc(1);
ccc_label_1:
    bar();
    if (ccc_mode == CCC_REWIND) return;
    ccc_pop_pc();
    ...
    ccc_push_pc(2);
ccc_label_2:
    #pragma omp baz
    {

```

```

        ...
    }
    if (ccc_mode == CCC_REWIND) return;
    ccc_pop_pc();
    ...
ccc_label_3:
    ccc_push_pc(3);
    #pragma omp parallel
    {
        parallel_body();
    }
    if (ccc_mode == CCC_REWIND) return;
    ccc_pop_pc();
    ...
}

void parallel_body()
{
    // Function with a single jump target.
    // Jump table, pushes and pops are omitted.
    if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
        goto ccc_label;
    ...
ccc_label:
    protocol_barrier();

```

```

    ...
}

void fib() {
    // Function with a single jump target.
    //Jump table, pushes and pops are omitted.
    if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
        goto ccc_label;
    ...
ccc_label:
    bat();
    if (ccc_mode == CCC_REWIND) return;
    ...
}

```

There is two crucial differences between the transformation described above and the equivalent transformation in in the sequential version of  $C^3$  [54].

First, this transformation also inserts statements,

```
if (ccc_mode == CCC_REWIND) return;
```

These are used to unwind the stack during checkpointing, which is required for recreating the stack after `critical` and `ordered` regions have been released as part of the checkpointing protocol.

Second, in sequential  $C^3$ , the PC stack was reconstructed on restart. In other words, the index of the next label was read from disk and the label pointed to the “`ccc_push_pc(...)`” call that pushes this label onto the PC stack. So the jump

table looked like this:

```

if (ccc_mode == CCC_RESTART) {
    switch (ccc_read_next_label()) {
        case 1: goto ccc_label_1;
        case 2: goto ccc_label_2;
    }
}

```

The OpenMP transformation assumes that the PC stack is read completely into memory before the user's `main()` function is called rather than read incrementally from disk. A variable `ccc_stack_cursor` is used to walk through the restored PC stack while the restart code is being executed.

The reason for the change is that to checkpoint `critical` and `ordered` regions we need to unwind portions of the stack and then recreate them. For this application it makes no sense to save the affected portions of the stack to disk and then read them back record-by-record. By reading PC stack records from memory as opposed to from the disk,  $C^3$  for OpenMP can use the same mechanism to recreate the entire stack on restart and to recreate portions of the stack during the checkpointing process (though, on restart  $C^3$  will first read the PC stack into memory from disk and then allow all subsequent reads to happen from memory).

### 3.8.2.6 Transformation #6: Stack Alignment

While the above mechanism successfully recreates all the regions that existed on the stack at the time of the checkpoint, there is no guarantee that the size of these regions will be the same on restart. For an example consider Figure 3.14(b), which shows the possible configuration of the stack on restart. Although it contains the

same regions: `main`, `#pragma omp parallel`, etc., the activation frames of application functions are mis-aligned relative to their locations at checkpoint time. As a result, application variables appear at different memory locations. This can cause incorrect restart behavior as the application may have pointers to the variables' old locations, which will be invalid on restart.

There are two ways in which this stack mis-alignment can happen. First, the stack itself may not appear in the same place as it did during the original execution. Second, the regions on the stack may not have the same size as they had at the time of the checkpoint.

#### 3.8.2.6.1 Stack Motion

The first problem can be dealt with in different ways, depending on how much the stack moves relative to the original execution. If the stack may only move by a small amount (several KB), this can be addressed by using `alloca` to pad the beginning of the stack with extra space. On restart, the amount of pad space allocated can be adjusted so that the application-visible portion of the stack begins at the same location. This needs to be done both at the top of `main` for the master thread and at the top of each `parallel` region for each non-master thread. The current  $C^3$  implementation uses this mechanism.

The following code,

```
void main() {  
    ...  
    #pragma omp parallel  
    {
```

```

        ...
    }
    ...
}

```

becomes

```

void main()
{
    int main_anchor_var;

    // during regular execution use default pad size
    if (ccc_mode == CCC_NORMAL)
        alloca(pad_buf_size);
    // on restart adjust the pad size to account for the new
    // stack start address
    else if (ccc_mode == CCC_RESTART)
        alloca(pad_buf_size +
               compute_stack_start_difference(&main_anchor_var));
    usr_main();
}

void usr_main()
{
    ...
    #pragma omp parallel
    {

```

```

int par_anchor_var;

// non-master threads start here and must align their
// stacks correctly
if(omp_get_thread_num()!=0)
{
    // during regular execution use default pad size
    if (ccc_mode == CCC_NORMAL)
        alloca(pad_buf_size);
    // on restart adjust the pad size to account for
    // the new stack start address
    else if (ccc_mode == CCC_RESTART)
        alloca(pad_buf_size +
            compute_stack_start_diff(&par_anchor_var));
}
...
}
...
}

```

The amount of adjustment needed is computed by using an anchor variable (`main_anchor_var` and `main_anchor_var` in the above pseudo-code) that is placed at the very top of each thread's stack and comparing the difference between its address during the original and on restart. Function `compute_stack_start_difference` performs the comparison for each thread and returns the amount of difference.

If the above assumption does not hold and on restart thread stacks may move by large amounts, it becomes necessary to Reimplement OpenMP's call stack functionality. In particular, it is possible to force each thread to use a new stack whose position is under the direct control of  $C^3$  by using `mmap` to allocate a new memory region and then using `setcontext` to force the thread to switch from its default stack to this new region. Windows has equivalent system calls. This technique is described in more detail in Section 3.2 of [139].

#### 3.8.2.6.2 Stack region size changes

The second reason for stack mis-alignment is that the regions on the stack may not have the same size as they had at the time of the checkpoint. The problem here is that while there is a guarantee that function activation frames will not change size between executions of the same executable binary, the same is not true of the amount of stack space taken up by OpenMP regions. In fact, the OpenMP specification says nothing about how much stack space a given OpenMP region takes up or whether this amount is even deterministic. Thus, on restart stack regions corresponding to OpenMP directives such as `#pragma omp parallel` or `#pragma omp critical` may be much larger or smaller than they were at checkpoint time.

To solve this problem, we can start by making the simplifying assumption that the amount of stack space taken up by an OpenMP region will not change by a large amount between different executions. This assumption should hold in general because there is little reason for the same OpenMP stack regions that should always carry roughly the same data to drastically change in size between two executions of the same program. Given this assumption it is possible to use

the same `alloca`-based stack alignment mechanism used to align the beginning of the stack. In order to align the start of that function's stack to its original address, `alloca` needs to be called inside every OpenMP directive that has a body. This call must be executed before calling the function that contains the lifted body of that directive. Since the body of every OpenMP directive needs to be aligned in this way, the individual stack padding regions must be made smaller (32-128 bytes). This is the mechanism currently used in *C*<sup>3</sup>.

If on some system the small-difference assumption does not hold, it is possible to use `mmap` and `setcontext` just like before, creating a new mini-stack for every portion of the call stack that belongs to the application. Thus, for the call stack in Figure 3.14, there would be a mini-stack for `main` and `#pragma omp parallel`, another for `foo` and `#pragma omp critical` and another for `bar`, thus ensuring that every application-controlled region of the stack appears at the same memory address on restart as during the original execution.

### 3.8.2.7 Transformation #7: Saving state of local and global variables

The transformations described above ensured that the stack of each thread is restored to contain the same regions as it did at checkpoint time and that all the application-controlled regions begin at their original addresses. However, the stack variables and global variables are still uninitialized and must be reloaded with their original data. The approach taken here is very similar to that of the sequential *C*<sup>3</sup> [54] [139](Section 4.2.2).

Global variables are restored to their original state by analyzing the application source code and identifying all global variables. *C*<sup>3</sup> then adds two functions to every source file: `chkpt_globals_id#` and `restore_globals_id#` where `id#` is the

unique id of the source file. `chkpt_globals_id#` saves the data of all the file's global variables to the checkpoint file and `restore_globals_id#` restores their values. Since these variables are global, they are guaranteed to be visible in these functions. Static local variables, which have global semantics but local scope, are checkpointed by  $C^3$  by pulling them outside their host functions (thus making them global) and giving them a unique prefix, thus preserving their local scope semantics. They are then checkpointed just like other global variables.

Local variables appear on and disappear from the call stack as they come into and out of scope.  $C^3$  uses the `svd_stack` (SVD = Stack Value Descriptor) to record information about all local variables on the stack at any given time, with one `svd_stack` per thread.  $C^3$  transforms the application by inserting after the entry into scope of a given variable a call to the function `ccc_push_svd`, which pushes onto the `svd_stack` the address and size of the new variable. A call to `ccc_pop_svd` is inserted whenever some variable leaves scope. As such, at all times the each thread's `svd_stack` contains the addresses and sizes of all local variables on the thread's call stack at that time. At checkpoint time, the `svd_stack` is checkpointed and used to save the data of these variables. On restart it is restored and used to restore the values of its thread's local variables after the thread's call stack has been recreated.

The process of maintaining the `svd_stack` can be optimized by restricting the insertion of `svd` function calls into just the code that may lead to a checkpoint. Furthermore, multiple calls to `ccc_push_svd` and `ccc_pop_svd` can be aggregated together if the application is transformed such that multiple variables enter scope at the same time. The details of these optimizations are not included here.

The following code,

```

int global;

void foo()
{
    static int staticLocal;
    int fooLocal;
    ... global ... staticLocal ... fooLocal ...
    potential_checkpoint();
    ... global ... staticLocal ... fooLocal ...
}

void main() {
    int mainLocal;
    ... global ... mainLocal ...;
    #pragma omp parallel
    {
        int parLocal;
        ... global ... mainLocal ... parLocal ...
        foo();
        ... global ... mainLocal ... parLocal ...
    }
    ... global ... mainLocal ...;
}

```

becomes (ignoring the other transformations)

```

int global;

```

```
int ccc_foo_staticLocal;

void foo() {
    int fooLocal;
    ccc_push_svd(&fooLocal, sizeof(int));
    ... global ... ccc_foo_staticLocal ... fooLocal ...
    potential_checkpoint();
    ... global ... ccc_foo_staticLocal ... fooLocal ...
    ccc_pop_svd();
}

void main() {
    int mainLocal;
    ccc_push_svd(&mainLocal, sizeof(int));
    ... global ... mainLocal ...;
# pragma omp parallel
{
    int parLocal;
    ccc_push_svd(&parLocal, sizeof(int));
    ... global ... mainLocal ... parLocal ...;
    foo();
    ... global ... mainLocal ... parLocal ...;
    ccc_pop_svd();
}
... global ... mainLocal ...;
```

```
        ccc_pop_svd();
    }

void chkpt_globals_0()
{
    chkpt_save(&global, sizeof(int));
    chkpt_save(&ccc_foo_staticLocal, sizeof(int));
}

void restore_globals_0()
{
    chkpt_restore(&global, sizeof(int));
    chkpt_restore(&ccc_foo_staticLocal, sizeof(int));
}
```

### 3.8.2.8 Transformation #8: Saving the state of the heap

In order to make it possible to ensure that all heap buffers are allocated on restart at their original locations  $C^3$  uses its own portable memory allocator implementation, documented in Section 3.3 of [139].  $C^3$  transforms the application to replace all calls to `malloc`, `free` etc. to their counterparts in this memory allocator.

### 3.8.3 Checkpointing Protocol

#### 3.8.3.1 Protocol state

The state of the protocol consists of a variety of shared and private variables.

##### 3.8.3.1.1 Shared variables

```
int checkpoint_initiated = 0;
```

This shared variable records the number of threads that have decided to take a checkpoint. It is used to indicate whether or not a checkpoint has been initiated by any thread and in this capacity is used as a boolean flag. In the case of checkpointing inside of `ordered` regions, it is used as a counter to tell threads when to stop skipping parallel `for` loop iterations. It incremented whenever a thread wishes to checkpoint and set to 0 at the end of each checkpoint.

```
LinkedList worksharing_records;
```

A linked list of records describing the worksharing tasks that have been allocated to the application's different threads. Managed by the function `record_workshare_assignment`.

```
int max_chkpt_workshare;
```

Each dynamically executed worksharing construct and region of code between worksharing constructs is assigned a unique id. On restart, `max_chkpt_workshare` holds the maximum id any thread had at checkpoint time.

```
bool threads_awoken[2] = {FALSE, FALSE};
```

At the time of a checkpoint some threads may be blocked on a barrier and unaware that other threads have decided to take a checkpoint and wish for them to follow. As a result, the checkpointing threads will call a barrier in order to wake the blocked threads up. After they have been awoken they will wish to know why they were awoken. Was it because all threads have also called a barrier or was it simply because other threads wanted to take a checkpoint? In order to communicate this, at checkpoint time before calling the "wakeup" barrier, the checkpointing threads will set the `threads_awoken` flag to TRUE. Two flags are used to avoid race conditions between the reads and writes associated with adjacent checkpoints and barriers. Each thread uses a `threadprivate` variable `barrier_id` in order to choose which flag to use. `barrier_id` is switched between 0 and 1 after every checkpoint and barrier.

### 3.8.3.1.2 Thread-private variables

```
int barrier_id = 0;
```

Used to switch between the two `threads_awoken` flags. `barrier_id` is switched between 0 and 1 after every checkpoint and barrier.

```
int ccc_mode = either CCC_NORMAL, CCC_RESTART, CCC_REWIND or
               CCC_REPLAY;
```

Executing code can be in one of four modes. The first two are also used in sequential C<sup>3</sup> [139] [54]. The other two are new to this protocol.

- `CCC_NORMAL`: Normal execution.

- `CCC_RESTART`: The application is being restarted. The application code is being executed in order to reconstruct the stack frames up to and including the call to `perform_checkpoint`.
- `CCC_REWIND`: The stack is being unwound in order to exit any `critical` or `ordered` regions.
- `CCC_REPLAY`: The application code is being reexecuted in order to reconstruct the stack after rewinding it to checkpoint inside of a `critical` or `ordered` region.

```
int ccc_workshare_mode = either CCC_NORMAL or CCC_RESTART;
```

Used to indicate whether we are currently executing worksharing constructs normally ( = `CCC_NORMAL`) or in a controlled mode used on restart to allow the application to finish the worksharing constructs which were only partially completed before the checkpoint ( = `CCC_RESTART`).

```
int cur_workshare=0;
```

Each dynamically executed worksharing construct and region of code between constructs is assigned a unique id (i.e. the first construct gets `id=0`, the next has `id=1`, etc.). `cur_workshare` holds the id of the current region that the thread is in.

```
int ccc_critical_depth = 0;
```

This private variable is used to count the number of `#pragma omp critical` regions that its application thread is currently nested inside of.

```
bool inside_ordered=FALSE;
```

Flag indicating whether this thread is currently executing inside an `#pragma omp ordered` region.

```
int *ccc_stack_cursor;
```

When `ccc_mode` is either `CCC_RESTART` or `CCC_REPLAY`, this variable is used to point to the next element within the PC stack that needs to be used to transfer control at the next jump table.

```
int ccc_parallel_depth = 0;
```

Counts the dynamic nesting of `#pragma omp parallel` regions. Currently, nested parallel regions are implemented as a single thread; this variable is used to control this behavior. Since there is a copy for each thread, each thread counts its own nesting depth.

### 3.8.3.2 Parallel regions

In this protocol, only the outermost parallel region generates new threads. Any inner parallel regions are actually single threaded. The following transformation shows how `parallel` regions are transformed and is an extension of the more generic transformation shown in Section 3.8.2.5. To make sure that the same number of threads is recreated as existed at checkpoint time, `omp_get_num_threads` is used to record the number of threads present at checkpoint time. `omp_set_num_threads` and `omp_set_dynamic` are used to ensure that the same number of threads is created on restart.

$C^3$  supports nested parallelism only to the extent of ensuring that each nested parallel region contains a single thread. This is done by calling `omp_set_nested` to disable the native OpenMP implementations support for nested parallelism and

replacing all application calls to `omp_set_nested` with calls to `ccc_set_nested`, which does nothing.

The following code,

```
void main() {  
    ...  
  
    #pragma omp parallel  
    {  
        ...  
        potential_checkpoint();  
        ...  
    }  
    ...  
}
```

becomes

```
void parallel_body();  
  
void main() {  
    // Ensure that nested parallelism is not used by OpenMP  
    omp_set_nested(0);  
  
    if (ccc_mode == CCC_RESTART)  
        goto ccc_label;  
    ...  
}
```

```

ccc_parallel_depth++;
ccc_label:
    // when recreating the threads make sure to recreate the
    // same number of threads
    if (ccc_mode == CCC_RESTART)
    {
        // disable dynamic decisions about the number of
        // threads in this region
        omp_set_dynamic(0);
        // request the same number of threads as existed at
        // checkpoint time
        omp_set_num_threads(read_chkpt_num_threads());
    }
    // The outermost parallel region is actually parallel.
    #pragma omp parallel
        parallel_body();
    if (ccc_mode == CCC_REWIND) return;
ccc_parallel_depth--;
    ccc_pop_pc();
    ...
}

void parallel_body() {
    if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
        switch (*ccc_stack_cursor++) {

```

```

    ...
    case i: goto ccc_label_i;
    ...
}
...
ccc_label_i:
    potential_checkpoint();
    if (ccc_mode == CCC_REWIND) return;
    ccc_pop_pc();
    ...
}

```

### 3.8.3.3 Master

**Master** regions are pieces of code executed only by the master thread. Since they provide no synchronization semantics,  $C^3$  only needs to do two things. At checkpoint time  $C^3$  records that a **master** region is on the call stack. The same **master** region is reentered on restart. Thus, **master** regions are only modified via Transformation #5 described in Section 3.8.2.5.

### 3.8.3.4 Atomic

**Atomic** regions are a specialized critical section that can only contain the update of some variable. These updates can be of the form  $x \oplus = expr$ ,  $x ++$ ,  $++x$ ,  $x --$  and  $--x$ , where  $\oplus$  is not overloaded and one of the following:  $+$ ,  $*$ ,  $-$ ,  $/$ ,  $\&$ ,  $\wedge$ ,  $|$ ,  $\ll$  or  $\gg$ . While atomic regions provide synchronization for the update operations, no synchronization is implied for the *expr* on the right hand

side of  $x \oplus = expr$ . As such, it is legal to extract the *expr* from the body of the `atomic` region, making it impossible for a potential checkpoint location to appear inside an `atomic` region and allowing  $C^3$  to ignore them for the purposes of other transformations.

Thus, the following code,

```
#pragma omp atomic
    x++;
#pragma omp atomic
    x*=foo();
```

becomes

```
#pragma omp atomic
    x++;
temp = foo();
#pragma omp atomic
    x*=temp;
```

### 3.8.3.5 Potential and explicit checkpoints

Calls to `potential_checkpoint` represent points in the code where it may be advantageous to take a checkpoint such as locations with small memory footprints. When `potential_checkpoint` is called it takes a checkpoint if (i) some other thread has decided to checkpoint or (ii) some external decision algorithm (encapsulated in function `time_to_checkpoint` decides that its time. `explicit_checkpoint` takes a checkpoint whenever it is called. Both functions call `perform_checkpoint`, which actually performs the checkpoint.

The pseudocode below enhances the version from Section 3.4.3 by providing full details about the work done during all the phases of an application's execution. If `perform_checkpoint` is called when `ccc_mode=CCC_NORMAL`, it atomically increments `checkpoint_initiated` and if the call is performed inside a `critical` or `ordered` region, it switches the mode to `CCC_REWIND` so that the call stack can be unwound and these resources released. If the thread does not hold such resources, `perform_checkpoint` continues execution by calling the first checkpoint barrier, releasing all other resources, recording application state, reacquiring locks and calling the second checkpoint barrier.

If the thread does hold `critical` or `ordered` regions, then the above algorithm is executed in parts. First, the thread sets `ccc_mode` to `CCC_REWIND` and returns, causing the thread to exit all surrounding `critical` or `ordered` regions. `perform_checkpoint` is called again (via `explicit_checkpoint`) and continues the checkpoint by calling the first checkpoint barrier and saving application state. It then sets `ccc_mode` to `CCC_REPLAY` and returns, causing the thread to reacquire previously held `critical` or `ordered` regions, after which time it calls `perform_checkpoint` again. This time `perform_checkpoint` finishes the checkpoint by reacquiring any locks it may have released, resetting `checkpoint_initiated` and calling the second checkpoint barrier.

Since OpenMP does not provide the guarantee of write atomicity, it is not possible to atomically increment `checkpoint_initiated` in a way that is guaranteed to work in all possible implementations of OpenMP (`#pragma omp atomic` is a type of critical section, not an atomic write). Fortunately, all real implementations of OpenMP provide their own mechanisms to enforce such atomicity. As such, this becomes a point where  $C^3$  goes beyond the OpenMP specification in

using the atomic write functionality of the OpenMP implementation that is not available in the OpenMP specification itself.

```

void potential_checkpoint() {
    if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
        goto ccc_label;

    if (checkpoint_initiated || time_to_checkpoint()) {
ccc_label:
        perform_checkpoint(FALSE);
        if (ccc_mode == CCC_REWIND) return;
    }
}

```

```

void explicit_checkpoint() {
    perform_checkpoint(FALSE);
}

```

```

void perform_checkpoint(bool at_barrier) {
    // perform_checkpoint() can be summarized with the
    // following pseudocode:
    // Atomic checkpoint_initiated++;
    // foreach c (criticals_held_by_current_thread())
    //     release_critical(c);
    // foreach o (orderedes_held_by_current_thread())
    //     release_ordered(o);
}

```

```
// foreach l (locks_held_by_current_thread())
//     release_lock(l);
// if (!at_barrier) release_barrier();
//
// global_barrier();
//
// if(master_thread) {
//     save_application_state();
//     save_hidden_state();
// }
//
// checkpoint_initiated = 0;
//
// foreach c (criticals_held_by_current_thread())
//     reacquire_critical(c);
// foreach o (ordereds_held_by_current_thread())
//     reacquire_ordered(o);
// foreach l (locks_held_by_current_thread())
//     reacquire_lock(l);
//
// global_barrier();

// Phase 1: If checkpointing inside critical or ordered
// regions, start unwinding the stack. This will
// release threads blocked at entry into critical or
```

```
//    ordered regions.
if (ccc_mode == CCC_NORMAL) {
    Atomic checkpoint_initiated++;
    if (ccc_critical_depth > 0 || inside_ordered) {
        ccc_mode = CCC_REWIND;
        return;
    }
}

// Phase 2: Release threads blocked at locks and barriers.
switch (ccc_mode) {
case CCC_NORMAL:
case CCC_REWIND: {
    foreach l (lock_held_by_current_thread())
        release_lock(&l);

    // The parameter "at_barrier" is used to indicate
    // whether or not the current thread was blocked at
    // a barrier prior to calling "ccc_perform_checkpoint".
    // If not, then it is necessary execute a barrier
    // in order to release any blocked threads on the
    // team. If so, then do not execute a barrier - the
    // current thread has already been released!
    if (!at_barrier)
        release_barrier();
}
}
```



```
case CCC_NORMAL:
case CCC_REWIND: {
    // first checkpoint barrier
    global_barrier();

    #pragma omp master
    {
        ccc_save_heap();
        ccc_save_gvd();
    }
    ccc_save_svd();

    break;
}

case CCC_REPLAY:
    break;
}

// Phase 4: If the stack was unwound, then it needs to
// be replayed.
if (ccc_mode == CCC_REWIND) {
    ccc_mode = CCC_REPLAY;
    // This always returns to the ccc_explicit_checkpoint
    // after the outermost critical/ordered region
```

```
        // that surrounds the original checkpoint location.
        // This will ensure that this critical/ordered
        // region is re-entered and the execution context
        // is reconstructed.
        return;
    }

    // Phase 5: Reinitialize the checkpoint flag.
    // Reacquire locks. Finish with a barrier.
    assert (ccc_mode != CCC_REWIND);
    // The thread is done CCC_REPLAY'ing or CCC_RESTART'ing.
    ccc_mode = CCC_NORMAL;
    #pragma omp master
    {
        checkpoint_initiated = 0;
        #pragma omp flush
    }
    foreach l (lock_held_by_current_thread())
        reacquire_lock(&l);
    if (!at_barrier) reset_barrier();

    // second checkpoint barrier
    global_barrier();
}
```

### 3.8.3.6 Locks

```
release_lock(omp_lock_t *l) {
    omp_unset_lock(l);
}

reacquire_lock(omp_lock_t *l) {
    omp_set_lock(l);
}

void protocol_set_lock(omp_lock *l) {
    omp_set_lock(*l);
    while (checkpoint_initiated) {
        omp_unset_lock(*l);
        perform_checkpoint(FALSE);
        omp_set_lock(*l);
    }
    record_resource_held(l);
}

int protocol_test_lock(omp_lock *l) {
    int acq_success = omp_test_lock(l);
    while (checkpoint_initiated) {
        omp_unset_lock(l);
        perform_checkpoint(FALSE);
    }
}
```

```
        acq_success=omp_test_lock(l);
    }
    record_resource_held(l);
    return acq_success;
}

void protocol_unset_lock(omp_lock_t *l) {
    remove_resource_held(l);
    omp_unset_lock(l);
}
```

### 3.8.3.7 Barriers

```
void forcerelease_barrier() {
    // Inform other threads that a checkpoint has begun
    threads_awoken[barrier_id] = TRUE;
    // Release threads that are blocked on barrier calls
    ccc_global_barrier();
}

void reset_barrier()
{
    // Reset the threads_awoken flag for the next
    // barrier/checkpoint
    threads_awoken[barrier_id] = FALSE;
}
```

```
// Switch to using the other threads_awoken flag for the
// next barrier/checkpoint
barrier_id = !barrier_id;
}

void protocol_barrier() {
    if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
        goto ccc_label;

    // if this thread is not inside a nested parallel region
    if(ccc_parallel_depth<=1)
    {
        // wait on a barrier, hoping that we will wake up only
        // because each thread's application has called a
        // barrier
        ccc_global_barrier();

        // check whether any thread has set the current
        // threads_awoken flag before calling its barrier
        while (threads_awoken[barrier_id])
        {
            // if some thread has, we need to immediately take
            // a checkpoint and then resume waiting on the
            // barrier again

            // Perform the checkpoint

```

```
        ccc_push_pc(1);
ccc_label:
        ccc_perform_checkpoint(TRUE);
        if (ccc_mode == CCC_REWIND) return;
ccc_pop_pc();
        // Reset the threads_awoken flag for the next
        // barrier/checkpoint
        threads_awoken[barrier_id] = FALSE;
        // Switch to using the other threads_awoken flag
        // for the next barrier/checkpoint
        barrier_id = !barrier_id;
        // Resume waiting on the barrier
        ccc_global_barrier();
    }
    // We now know that on all threads the application
    // has called a barrier, meaning that the barrier
    // has been completed

    // Switch to using the other threads_awoken flag for
    // the next barrier/checkpoint
    barrier_id = !barrier_id;
}
// else, if this thread is inside a nested parallel region
else
{
```

```

        // Since the nested parallel region is running with
        //   a single thread, a barrier call is a noop since
        //   the thread will be synchronizing with itself.
        // As such, do nothing since there is not even a reason
        //   to force a checkpoint here.
    }
}

```

As part of the protocol,  $C^3$  provides its own implementation of barriers (`ccc_global_barrier`) that is used instead of the native OpenMP barrier. The reason why this is done is because the OpenMP spec prohibits OpenMP barriers from appearing within certain constructs, such as critical regions and worksharing constructs. As a result, we use our own implementation of barrier to avoid a violation. Experimental results (Section 3.9.3.1) show that the overhead of doing so is relatively small and it is easily possible to provide for each platform a specially tuned barrier implementation.

```

void ccc_global_barrier() {
    // Insert your favorite barrier implementation here...
}

```

### 3.8.3.8 Critical regions

The fundamental problem with critical regions is that in order to release any blocked threads, a thread checkpointing inside of a region section must exit the critical region first. This results in a protocol that rewinds and replays parts of the execution stack.

Here is what happens for a thread that checkpoints inside of a critical region.

- Thread enters critical region with `ccc_mode == CCC_NORMAL`.
- Thread calls `perform_checkpoint` from inside of some function such as `potential_checkpoint` or `protocol_barrier`.
- Since `ccc_critical_depth > 0`, the thread is in at least one critical region. `ccc_mode` is set to `CCC_REWIND` and return is executed. The code inserted by transformation #5 (Section 3.8.2.5) will ensure that all enclosing routines are exited until the outermost critical region is reached.
- The thread calls `perform_checkpoint` again. Locks and barriers are released. All of the state is saved. `ccc_mode` is set to `CCC_REPLAY`.
- The thread re-enters the critical region. The execution context is reconstructed in much the same way as it would be on restart.
- After the thread's stack is recreated, it reenters `perform_checkpoint` and finishes the checkpointing process. All locks are reacquired. The final barrier is executed.

Thus, the following code

```
#pragma omp critical
{
    ...
}
```

becomes,

```
if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
    switch (*ccc_stack_cursor++) {
```

```
    case 1: goto ccc_label_1;
    case 2: goto ccc_label_2;
}

// Critical regions outside of parallel regions are not allowed.
assert(ccc_parallel_depth > 0);

int saved_stack_top = ccc_top_pc_stack();

while (TRUE) {
    ccc_push_pc(1);
ccc_label_1:
    // If a thread is forced to take a checkpoint while blocked
    //   on this critical section, it will try to restart
    //   inside the body of the if statement below. Since this
    //   is the only time when its body will be executed, this
    //   if has a false entry condition.
    if(0) {
ccc_label_2:
        ccc_explicit_checkpoint();
    }
#pragma omp critical
{
    // If checkpoint_initiated is true, then one of two
    // conditions holds,
```

```
// - The thread entered the critical region because
// another thread left it in order for the checkpoint
// protocol to proceed. Do not execute anything;
// leave the critical region immediately.
// - The application is in the middle of a checkpoint
// and this thread started its checkpoint inside of
// a critical region. The thread executed a series
// of returns in order to release the critical region.
// Now it must CCC_REPLAY the creation of the
// execution context.
if (!checkpoint_initiated || ccc_mode == CCC_REPLAY) {
    ccc_critical_depth++;

    // Execute the application code inside the critical
    // region
    ...

    ccc_critical_depth--;
}
} // end of #pragma omp critical
// If the thread did not take a checkpoint inside the above
// critical region
if (ccc_mode == CCC_NORMAL) {
    // If the thread normally exited the above critical
    // region
```

```
if (!checkpoint_initiated) {
    // continue executing normally
    ccc_pop_pc();
    break;
}

// Else, if we are here because the thread entered the
//   above critical region in CCC_NORMAL mode and was
//   forced to take a checkpoint.
else {
    // Begin the checkpoint. However, since this thread
    //   was forced to checkpoint immediately before
    //   it was able to enter the critical section,
    //   the checkpoint should look like it was taken
    //   at that location. So set the pc stack up to
    //   reflect this fact before checkpointing.
    ccc_pop_pc(); // pop the 1 from the pc stack
    ccc_push_pc(2);
    ccc_explicit_checkpoint();
    if (ccc_mode == CCC_REWIND) return;
    // If we are outside of any critical or ordered
    //   regions, the above call to
    //   ccc_explicit_checkpoint() must have
    //   completed the entire checkpoint. As such,
    //   we are done and all that's left is to jump
    //   to immediately before the critical region
```

```
        //    and try entering it again.
        goto ccc_label_1;
    }
}

// Else, if the thread did take a checkpoint inside the
//    above critical region and it is in the middle on
//    rewinding its call stack to exit any surrounding
//    critical and ordered regions.
else if (ccc_mode == CCC_REWIND) {
    // If this is not the outermost critical region,
    // continue unwinding the stack.
    if (ccc_critical_depth > 1) return;
    // If we reach this point, then this must be the
    // outermost critical region. Continue the checkpoint.
    ccc_explicit_checkpoint();
    // Start CCC_REPLAY'ing the execution context.
    assert(ccc_mode == CCC_REPLAY);
    // Set the stack cursor.
    ccc_stack_cursor = saved_stack_top;
    // go to the entry point of the critical region
    // and try to reacquire it
    goto ccc_label_1;
}
}
```

### 3.8.4 Worksharing Constructs

OpenMP worksharing constructs are a mechanism for assigning work (pieces of code that need to be executed) to different threads. OpenMP provides three mechanisms for identifying pieces of code as chunks of work to be assigned in this fashion: `single`, `sections` and `loop`.

- The `single` construct identifies a piece of code and indicates that any thread may execute it.
- The `sections` construct identifies several pieces of code (called sections) and indicates that any thread may execute any of them (as long as they are all ultimately executed).
- The `loop` construct is a for-loop where each iteration may be executed by any thread. Furthermore, for each iteration assigned, the loop iteration variable will be set to the appropriate value for that iteration. For example, in the following code there are 200 work chunks, one for each iteration of the loop. If a given thread is assigned the first work chunk, it will execute the loop body with the value of  $i=0$ . If another thread is given the 10th iteration, its value of  $i$  will be 50, etc. The loop iterator variable is automatically made private for the duration of the loop in order to allow multiple threads to concurrently execute different loop iterations.

```
#pragma omp for
for(i=0; i<1000; i+=5)
{ ... loop body ... }
```

There are significant restrictions on the syntax of the for loop iteration conditions.

- The initial and final values of the iterator variable must be constant and the same on all threads
- The iterator variable may only be updated using an additive operation (+ or -) and the step size must also be constant and the same on all threads.

The result is that the set of values of the iterator variable must be describable using a linear expression (linear:  $y=ax+b$ ).

OpenMP has an additional restriction that requires every thread in an application to pass through the same sequence of barriers and worksharing constructs.

OpenMP worksharing constructs have a certain amount of hidden state that must be restored on restart. This state includes:

1. The work chunks that have already been executed, and
2. The work chunks currently being executed (if they are) by each thread.

### **Basic Worksharing Constructs**

$C^3$  needs to restore this state. Unfortunately, OpenMP's worksharing constructs provide limited facilities for this. In particular, it is not possible to force OpenMP to assign a given work chunk to the thread that was working on it at checkpoint time (in general, the choice of which work chunk is assigned to which thread can be influenced by the application but not forced). As a result, in order to allow a thread to resume working on the work chunk that it had at checkpoint time, we need to do the following.

- Extract the code of the work chunk and place it before the original work-sharing construct.
- On restart, when a thread needs to resume executing within the work chunk, it simply jumps directly into the extracted copy of its code and computes there. Since we will be restoring the rest of the thread's state to its original configuration (including the value of the iterator variable if this is a `loop` construct), the application will not be able to tell the difference between executing inside the copy of the work chunk code inside the worksharing construct and the copy created by the preprocessor.
- Because `loop` constructs automatically privatize the loop iteration variable, the same needs to be done for their extracted bodies. As such,  $C^3$  performs the privatization transformation of Section 3.8.2.4, where it extracts the body of the `loop` construct into a function and passes the iterator variable by value into this function.

Shown below is the above loop example transformed to allow a thread to restart inside the iteration that it took a checkpoint in.

```

if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
    switch (*ccc_stack_cursor++) {
        ...
        case j: goto ccc_label_j;
        ...
    }
    ...
    ccc_push_pc(j);

```

```

ccc_label_j:
    // if we are still trying to recreate the call stack upto
    // the point where the checkpoint was taken
    if(ccc_mode == CCC_RESTART)
    {
        loop_body(iter_at_checkpoint(thread_id));
    }

    ... resume the #pragma omp for loop...

void loop_body(int i) {
    ...
    loop body
    ...
}

```

On restart a thread will jump to the point just before the worksharing construct and execute the remaining code in the extracted copy of the for loop's body. Its iterator variable `i` will be restored to the value it had at checkpoint time. The thread may then take more checkpoints inside the for loop's body, as it wishes. When the construct body is finished, the next action that is performed depends on the type of construct. If this is a `single` construct, then we are done. In the case of `sections` and `loop`, the thread must resume executing the `#pragma omp sections` or `#pragma omp for` in order to execute any work chunks that were not yet assigned by OpenMP at the time of the checkpoint.

### Executing Remaining Work Chunks

The difficulty with executing the remaining work chunks on restart is that given the same `#pragma omp sections` or `#pragma omp for` as before, OpenMP will reexecute all of its work chunks from start, as if none of them have been executed. One solution would be to allow OpenMP to allocate previously executed work chunks to threads but to then have threads skip over the code of such work chunks using `goto`. While effective, this would mean that threads would need to skip a potentially large number of work chunks, causing restarts to take time proportional to the number of work chunks already executed by checkpoint time. Although this is acceptable for `sections` constructs, it will be quite expensive for parallel `loops`, which may have large numbers of iterations (for example, the EP code in the NAS Parallel Benchmarks [17] [18] spends almost all of its time in a single parallel `for` loop).

A more efficient alternative is to use an indirection array to only execute the iterations that were not completed by the time of the checkpoint. In particular, we can set the variable `num_remaining_iters` to the number of unassigned iterations remaining in the `for` loop and execute that many iterations. We can also set the array `remaining_iters` to contain the values of the iterator variable for each remaining iteration. Then, instead running the `for` loop over the original set of iterations, on restart we will run it over the set of remaining iterations. We extend the above transformed code to include this piece of the transformation.

```

if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
    switch (*ccc_stack_cursor++) {
        ...

```

```

case j: goto ccc_label_j;
...
}
...
ccc_push_pc(j);
ccc_label_j:
// if we are still trying to recreate the call stack upto
// the point where the checkpoint was taken
if(ccc_mode == CCC_RESTART)
{
    loop_body(iter_at_checkpoint(thread_id));

// now that we are done executing the iteration that
// we checkpointed in, assign the remaining loop
// iterations among threads
remaining_iters[*] =
    condense_remaining_iters();
#pragma omp for private(i)
for (ii=0; ii<num_remaining_iters; ii++) {
    record_workshare_assignment(cur_workshare,
                                remaining_iters[ii]);
    loop_body(remaining_iters[ii]);
}
}
// otherwise, if this is a regular execution then execute

```

```

// the worksharing construct normally, while also
// recording which work chunks have been executed
else if(ccc_mode == CCC_NORMAL)
{
    #pragma omp for
    for(i=0; i<1000; i+=5)
    {
        record_workshare_assignment(cur_workshare, i);
        ...
        loop body
        ...
    }
}

void loop_body(int i)
{
    ...
    loop body
    ...
}

```

If a thread checkpointed inside a worksharing construct, on restart it will jump into the copy of the construct's body. After it is done executing the body it will set `remaining_iters` to contain the iterator values for all the iterations that remain to be assigned. It then calls `#pragma omp for` to iterate over all the entries of `remaining_iters`, thus assigning all the remaining iterations to threads. In the

above code the function `record_workshare_assignment` keeps track of which work chunks have been assigned to each thread. `cur_workshare` is a private variable that uniquely identifies the dynamic instance of each given worksharing construct. It is described in more detail below.

The default behavior for OpenMP worksharing constructs is to have a barrier immediately following them. If this is not the behavior desired by the programmer, the `no_wait` flag can be used to designate a given worksharing construct to not be followed by a barrier. However, if `no_wait` flag is specified for a worksharing construct, some threads will be allowed to execute past the construct and potentially enter other worksharing constructs before other threads will have finished with the original construct.

The checkpointing solutions for the `single` and `sections` worksharing constructs are similar to those used for `loop` constructs since all types of worksharing constructs are simply different types of syntactic sugar for the same work scheduling mechanism. These transformations are omitted here and in the rest of this sub-section because they merely use simple variants of the transformations discussed here.

### **Issues with `no_wait`**

If all the worksharing constructs in an application do not have `no_wait` specified then it will not be possible for one thread to take a checkpoint in one worksharing construct while another is taking a checkpoint in a different construct and the above solution will work. However, if the program does have some `no_wait` worksharing constructs then the above solution fails, which presents us with new challenges.

The problem with the above solution can be seen in the example in Figure 3.15.

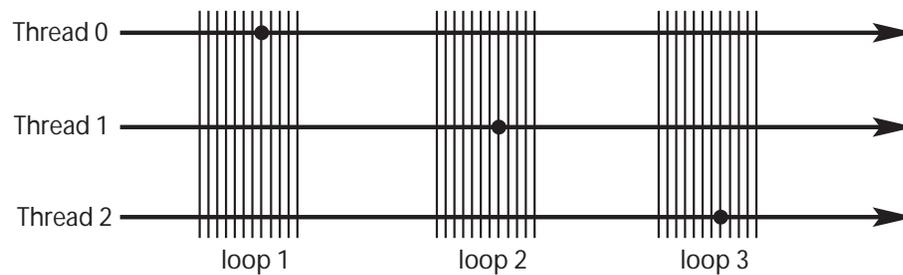


Figure 3.15: Sample execution with `no_wait` parallel loops

In this Figure thread 0 takes a checkpoint in an iteration of the first loop, thread 1 checkpoints inside the second loop while thread 2 checkpoints inside the third loop. On restart the above mechanisms will ensure that each thread restarts inside the iteration that it checkpointed in. However, after each thread finishes working on this work chunk we will have a problem. OpenMP requires that every thread encounter the same sequence of barriers and worksharing constructs during its execution. However, since on restart different threads will be executing in different worksharing constructs, the above transform will result in different threads calling a different sequence of worksharing constructs.

In particular, when thread 0 finishes working on the iteration that it took a checkpoint in, it will try to use `#pragma omp for` to assign the remaining unassigned iterations. However, threads 1 and 2 will restart at a location in the program where they have already passed the first parallel loop and will not be able to call it. In particular, if they do try to call a `#pragma omp for` that is identical to the one issued by thread 0, they risk being assigned some iterations from that loop, which would be erroneous since the application on those threads has already restarted in

a state where it believes that they have already passed it.

Thus, the only way for thread 0 to assign the unassigned iterations of the first loop is to do it without using `#pragma omp for`. This requires  $C^3$  to have its own work chunk scheduler algorithm and implementation. This algorithm will likely not be as good as those employed by native OpenMP implementations. However, it is only necessary to use the  $C^3$  implementation of work chunk assignment for worksharing constructs where some thread took a checkpoint after it left them. Given that the checkpointing protocol is blocking, if one thread takes a checkpoint in one construct, it is unlikely for other threads to make it very far before they take a checkpoint themselves. As such, the  $C^3$  work chunk scheduler implementation will be used for only short periods of time during restarts. The code below shows the final transformation of the above parallel loop that takes into account the fact that threads may checkpoint in different worksharing constructs.

```

    if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
        switch (*ccc_stack_cursor++) {
            ...
            case j: goto ccc_label_j;
            ...
        }
    ...
    ccc_push_pc(j);
ccc_label_j:
    cur_workshare++;

    // if we are still trying to recreate the call stack upto

```

```
// the point where the checkpoint was taken
if(ccc_mode == CCC_RESTART)
{
    loop_body(iter_at_checkpoint(thread_id));
}

// if we are past the last checkpointing construct that
// any thread took a checkpoint in, then we are done
// performing a controlled execution of worksharing
// constructs and may now get back to the regular
// execution
if(cur_workshare>max_chkpt_workshare)
    ccc_workshare_mode = CCC_NORMAL;

if(ccc_workshare_mode == CCC_NORMAL)
{
    #pragma omp for
    for(i=0; i<1000; i+=5)
    {
        record_workshare_assignment(cur_workshare, i);
        ...
        loop body
        ...
    }
}
```

```
else if(ccc_workshare_mode == CCC_RESTART)
{
    // if we are at a worksharing construct that some
    // threads had already passed by the time they took
    // a checkpoint
    if(cur_workshare<max_chkpt_workshare)
    {
        foreach(iteration i that has not yet been assigned)
        {
            record_workshare_assignment(i);
            loop_body(i);
        }
    }
    // else, if we are at the last checkpointing construct
    // that any thread took acheckpoint in
    else if(cur_workshare==max_chkpt_workshare)
    {
        // use the OpenMP parallel loop constructs to
        // schedule the remaining iterations
        remaining_iters[*] =
            condense_remaining_iters();
        #pragma omp for
        for (ii=0; ii<num_remaining_iters; ii++) {
            record_workshare_assignment(cur_workshare,
                remaining_iters[ii]);
        }
    }
}
```

```

        loop_body(remaining_iters[ii]);
    }
}

cur_workshare++;

void loop_body(int i) {
    ...
    loop body
    ...
}

```

In the above example `cur_workshare` is a private variable keeps track of which worksharing construct each thread is currently executing (i.e. 1st, 2nd, 3rd, etc); if it is not currently executing in a worksharing construct, the variable keeps track of the current inter-worksharing construct region of code. It is incremented immediately before and immediately after each worksharing construct. `max_chkpt_workshare` is a shared variable computed on restart that holds the maximum value of `cur_workshare` among all threads at checkpoint time. When a thread needs to assign the unassigned work chunks of a worksharing construct such that there exist some threads that completed this construct at the time of the checkpoint, (i.e. if  $\text{cur\_workshare} \leq \text{max\_chkpt\_workshare}$ ) then it uses the  $C^3$  implementation of work chunk assignment to assign the remaining work chunks. However, if  $\text{cur\_workshare} = \text{max\_chkpt\_workshare}$  then a native OpenMP work-sharing directive may be used to reassign the remaining work chunks since every

thread is able to call such a directive.

Finally, `ccc_workshare_mode` is a private flag that is set to `CCC_RESTART` as long as the thread still has not passed the last worksharing construct or inter-construct region that some thread checkpointed in. In this case  $C^3$  uses the techniques discussed above to assign the remaining work chunks. However, when `cur_workshare > max_chkpt_workshare` it is known that all worksharing constructs encountered from this point on will be executed from scratch by all threads, meaning that we can simply call the original worksharing construct as specified in the application. As such, `ccc_workshare_mode` is set to `CCC_NORMAL` to record this fact.

### Worksharing and Nested Parallelism

The final issue associated with worksharing constructs is their behavior inside nested parallel regions. Since a worksharing construct may be executed inside a nested `parallel` region that itself was executed inside of a `critical` or `ordered` region, it may be necessary to exit the this worksharing construct in order to release any relevant resources and then reenter it. While Transformation #5 in Section 3.8.2.5 provides the appropriate record keeping and stack unrolling and recreation functionality, this is not sufficient in the case of worksharing constructs since the only way to exit a worksharing construct is to execute or skip over each of its iterations. Since the total number of iterations may be large this may be prohibitively expensive.

The  $C^3$  solution is to observe that since it only supports nested parallel regions that contain only a single thread, a worksharing construct in such a region may only be executed by a single thread. As a result, it is possible to replace such

worksharing constructs with serialized versions of themselves without losing any performance. Exiting these serialized versions of the worksharing constructs becomes trivial and may be done using a simple `break` statement. Thus, the above transformations can be extended to the case of nested parallelism as follows:

```

if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
    switch (*ccc_stack_cursor++) {
        ...
        case j: goto ccc_label_j;
        ...
    }
...
    ccc_push_pc(j);
ccc_label_j:
    cur_workshare++;

// if this thread is not inside a nested parallel region
if(ccc_parallel_depth<=1)
{
    ... code of non-nested worksharing ...
    ...      construct from above      ...
}

// else, if this thread is inside a nested parallel region
else
{
    // the initial value of the loop iterator

```

```
int iter_start;

if(ccc_mode==CCC_NORMAL)
    // during normal execution use the default
    // initial value
    iter_start=0;
// else, if this thread is recreating its call stack
else
    // use the iteration number of the loop iteration
    // inside which a checkpoint was taken
    iter_start = get_last_iter(cur_workshare);

// execute the serialized loop
for(i=iter_start; i<1000; i+=5)
{
    record_workshare_assignment(cur_workshare, i);
    loop_body(i);
    // if loop_body was exited because this thread
    // began taking a checkpoint
    if(ccc_mode == CCC_REWIND)
        break; // exit the for loop
}
}

cur_workshare++;
```

```
void loop_body(int i) {  
    ...  
    loop body  
    ...  
}
```

#### 3.8.4.1 Ordered

The OpenMP `ordered` construct can be used inside of parallel for loops (`#pragma omp for`) to identify a piece of code as a type of critical section. The difference is that while critical sections only guarantee mutual exclusion, `ordered` regions are also guaranteed to be executed in the same order as the iterations that they are inside of. In particular, if a given iteration of a given parallel for loop contains an `ordered` region (not all have to), the code in the region is guaranteed to execute before the `ordered` regions of any later iterations. A parallel for loop that may contain `ordered` regions must be annotated with the `ordered` clause. Any iterations inside such an annotated a loop may contain `ordered` regions but may not execute more than one `ordered` region in any one iteration.

Since entry into an `ordered` region is a blocking operation, the checkpointing protocol detailed in Section 3.8.3.5 needs to be able to unblock any thread that is waiting to enter an `ordered` region before proceeding with the checkpoint. The only way for an application-level solution to unblock a thread waiting on a resource is to give this thread the resource and ensure that after it acquires the resource the thread will realize that it should not have gotten the resource and will immediately take a checkpoint. While this is easily done for locks and not quite so easily for

critical regions, it is more difficult to unblock a thread that is waiting to enter an `ordered` region.

```
#pragma omp for ordered
for(i=0; i<1000; i+=5)
{
    ... loop body ...
    ...
    #pragma omp ordered
    {
        ... ordered body ...
    }
    ...
}
```

Suppose threads 0 and 1 are both executing inside the parallel for loop above and thread 0 decides to take a checkpoint inside iteration 0 ( $i=0$ ). At the same time thread 1 has been assigned iteration 5 ( $i=25$ ) to execute, which it does until it reaches an `ordered` construct. At this point thread 1 blocks and waits for thread 0 to execute upto the end of the `ordered` region in iteration 4. In order to force thread 1 to participate in the checkpoint, thread 0 needs to allow it to enter its `ordered` region. However, since thread 0 is trying to take a checkpoint, it will not complete iterations 0 through 4 until it is finished with its checkpoint. In order for thread 0 to allow thread 1 to enter its `ordered` region, it must (i) exit its `ordered` region and (ii) skip over iterations 0 through 4 in order to allow thread 1 to enter its `ordered` region in iteration 5. While skipping these iterations thread 0 will

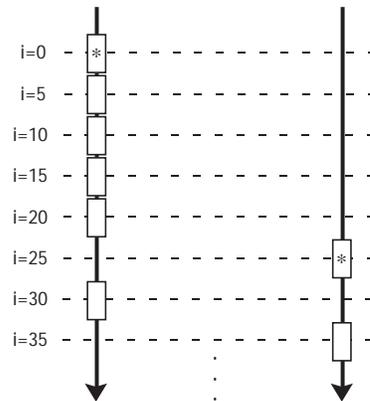


Figure 3.16: Sample execution of parallel for loop with `ordered`.

need to record which iterations were skipped so that it can execute their code after the checkpoint is completed.

This situation can be seen in Figure 3.16 and the algorithm for checkpointing it appears in Figure 3.17.

In these Figures each horizontal dashed line represents an iteration of the loop. Each box corresponds to a given thread being allocated a given iteration. In particular, thread 0 is given iterations  $i=\{0, 5, 10, 15, 20, 30, \dots\}$  while thread 1 gets the  $i=\{25, 35, \dots\}$  (this assignment is not known to the threads until they actually execute these iterations). The \*'s identify the iteration that each thread is currently in.

If thread 0 wants to checkpoint, it will jump to the end of its current iteration, exiting any `ordered` regions and `critical` sections that it may be inside of. It will then skip any iterations that thread 1 may be waiting on if it is blocked on entry into an ordered region. If any thread  $t$  is known to have entered iteration  $j$  of the loop, threads that wish to checkpoint will need to skip over all iterations upto and including  $j-1$  in case  $t$  is blocked on entry into iteration  $j$ 's ordered region. In the

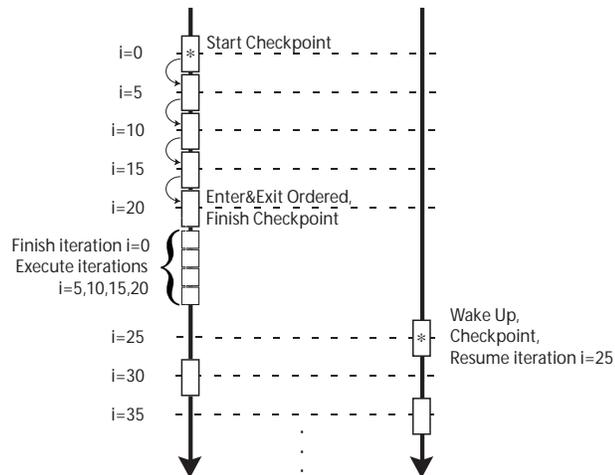


Figure 3.17: Example execution of `ordered` checkpointing algorithm.

example above, thread 0 will need to skip iterations until it reaches and skips over iteration 5 ( $i=20$ ), allowing thread 1 to wake up and take a checkpoint, as shown in Figure 3.17.

Once the checkpoint has been completed, thread 0 will have to finish up executing the iteration where it originally took a checkpoint and execute any iterations that it skipped. While doing so, it cannot use the native OpenMP implementation of `ordered` because:

- The OpenMP implementation will believe that the skipped iterations have been executed and will therefore not be able to provide appropriate synchronization semantics.
- While skipping iterations some threads may wind up skipping past of the end of the parallel for loop. Such threads will still need to execute the iterations that they skipped but will not be able to use the `ordered` construct, since it is only available inside parallel for loops.

As such, during this phase of iteration execution  $C^3$  uses its own implementation

of the `ordered` synchronization. This results in some iterations using  $C^3$ 's implementation of `ordered` while others use OpenMP's implementation. To prevent this discrepancy from causing synchronization errors, the first `ordered` region executed in a non-skipped iteration after the checkpoint uses both the  $C^3$  and the OpenMP implementations of `ordered` in order to properly synchronize it relative to both the preceding iterations, which use  $C^3$ 's `ordered` implementations, and the following iterations, which use OpenMP's native implementation of `ordered`.

It is possible for a checkpoint to be taken while skipped iterations are being executed. As such, the iteration re-execution code must itself have a form that is very similar to a transformed `ordered` region to enable it to be self-checkpointing. If a checkpoint is taken inside a skipped iteration, the fact that skipped iterations use  $C^3$ 's implementation of the ordered synchronization means there is no need to re-skip the remaining skipped iterations as would be done with regular iterations. Because this implementation of `ordered` is under the control of  $C^3$ , it exports function `ccc_ordered_nonblocking`, which turns off blocking on entry into a  $C^3$  `ordered` region. `ccc_ordered_blocking` is used to reset the blocking behavior once it is known that all threads have begun the checkpoint.

### **Termination of iteration skipping**

The iteration skipping process described above must proceed until the skipping threads are certain that there does not exist a thread that is executing inside a later iteration. However, it does not provide a way for threads to decide when to stop skipping.

One option is to use the fact that `checkpoint_initiated` contains the number of threads that have begun checkpointing. Threads can continue skipping iterations until `checkpoint_initiated` is equal to the number of threads in the

application because that is proof no thread is being prevented from checkpointing because it is blocked on a resource. This mechanism is efficient during the application's regular execution because it does not require any additional synchronization among threads. During checkpointing, however, it can be expensive because it is quite conservative, continuing to skip iterations even if there is no chance for any thread to become blocked on an `ordered` region. As such, it may skip many more iterations that are strictly necessary.

The solution is to have threads maintain a shared `ccc_max_iter` array of counters (one entry per thread) that keeps track of the maximum iteration that has been reached by each thread (this information is obtainable from the data structures maintained by `record_workshare_assignment` but is shown separately here for clarity). A thread that has begun its checkpoint skips a given iteration `j` if `max_chkpt_workshare > cur_workshare` (meaning that some thread has already finished the `for` loop) or `j < ccc_max_iter[t]` for some thread `t` (meaning that thread `t` is currently executing in a later iteration). This process is guaranteed to skip no more iterations than necessary, assuming that all iterations contain `ordered` regions. It is conservative in the case where some iterations do not have `ordered` regions. It is also guaranteed to terminate because eventually every thread will either block on the entry into an `ordered` region or take a checkpoint (we have assumed that potential checkpoint locations are placed such that they are reached regularly).

Note that since this approach uses synchronization through variables, it assumes that writes are atomic. While this is true in almost every implementation of OpenMP (at least for 32-bit writes), it is not guaranteed by the OpenMP specification [52]. As such, this aspect of  $C^3$  is implementation-specific and may need

to be slightly adjusted in OpenMP implementations with different write atomicity guarantees.

### **Restarting in the presence of ordered regions**

Since the protocol is blocking, it can only checkpoint the state of the application's threads as it was at some actual point in time. As such, on restart it is not necessary to perform any skipping and all the techniques described in Section 3.8.4 are sufficient for restarting applications that use `ordered` regions. The only difference is that on restart there will be some parallel `for` loops for which it will not be possible to use the native `#pragma omp for` and  $C^3$  will need to assign work chunks on its own, as discussed in Section 3.8.4 above. In order to enforce proper synchronization between the `ordered` regions inside such iterations  $C^3$  uses its own implementation of `ordered` rather than the one provided by OpenMP. Since on restart the work chunks of any given worksharing construct will be assigned using either  $C^3$ 's scheduler or OpenMP's native scheduler, with no mixture cases, there is no need to worry about transitioning from using our implementation of `ordered` to OpenMP's implementation.

The above protocol ideas lead to the parallel `for` loop above being transformed into the protocol pseudo-code below (pseudo-code for restarting the worksharing construct is omitted since this was covered in Section 3.8.4 above).

```
// after checkpointing, the last iteration for which the
// application will be using C3's implementation of
// ordered synchronization
shared int max_iter_ccc_ordered=-1;

// the maximum iteration number reached by each thread in its
```

```
// current parallel for loop
shared int ccc_max_iter[omp_get_num_threads()] = 0;

if (ccc_mode == CCC_RESTART)
    switch (*ccc_stack_cursor++) {
        ...
        case j: goto ccc_label_j;
        ...
    }

    ccc_push_pc(j);
ccc_label_j:
#pragma omp for ordered
for(i=0; i<1000; i+=5)
{
    // if this is a regular execution
    if(ccc_mode == CCC_NORMAL)
    {
        // record that this thread was assigned iteration i
        record_workshare_assignment(cur_workshare, i);

        loop_body(i);

        // If a checkpoint was taken inside the loop
        // (i.e. ccc_mode==CCC_REWIND), this iteration will
        // end and subsequent iterations will begin
    }
}
```

```
        // iteration skipping in order to release threads
        // blocked on entry to OpenMP-implemented ordered
        // regions
    }

    // else, if this thread is in the middle of a checkpoint
    // and still busy skipping iterations
    else if(ccc_mode==CCC_REWIND)
    {
        // determine if any more iterations need to be skipped
        bool skip = rewind_loop_body();

        // If they do not need to be skipped, finish the
        // checkpoint by recreating the call stack of of the
        // iteration inside which this thread took a
        // checkpoint and execute all the iterations
        // skipped by this thread
        if(!skip)
            replay_loop_body();
    }
} // end of for(i=0; i<1000; i+=5)

// If the thread exited the parallel for loop while
// ccc_mode==CCC_REWIND then then it must have been
// skipping iterations to release other threads for
// checkpointing.
// This thread has no more skipping to do (it has nothing to
```

```
// skip) and must thus do the restart portion of the
// checkpoint:
// - recreate the call stack of the iteration where the
// checkpoint was taken,
// - finish the checkpoint and the rest of that iteration,
// - execute iterations skipped by this thread.
if(ccc_mode==CCC_REWIND) {
    // At this point this thread cannot skip any more
    // iterations It now needs to finish up the checkpoint
    // by reentering the ordered construct inside/before
    // which this thread took its checkpoint. It must also
    // execute all the iterations skipped by this thread.
    replay_loop_body();

    // If a checkpoint was taken while executing one of the
    // skipped iterations, this thread has no iterations to
    // skip since it has already finished with the parallel
    // for loop and replay_loop_body() must have already
    // released threads blocked on entry into C3 ordered
    // regions by calling ccc_ordered_nonblocking().

    while(ccc_mode==CCC_REWIND) {
        // thus, participate with other threads in completing
        // the iteration skipping process
        rewind_loop_body();
    }
}
```

```
    // finish this checkpoint and resume executing skipped
    // iterations
    replay_loop_body();

    // Repeat the above every single time a checkpoint is
    //   taken while this thread is executing a skipped
    //   iteration until all skipped iterations have been
    //   executed
}
}

// Performs the rewind portion of a checkpoint, skipping any
//   iterations that another thread's ordered region may
//   depend on.
// Returns TRUE if the current iteration must be skipped and
//   FALSE if this thread is done skipping
bool rewind_loop_body() {
    // stall until we can decide whether to skip this
    // iteration or to finish this checkpoint
    while(checkpoint_initiated < omp_get_num_threads())
    {
        #pragma omp flush (ccc_max_iter)

        // determine the maximum iteration that has been
        // entered by any thread
    }
}
```

```
int max_iter=-1;
for(int thread=0; thread<omp_get_num_threads();
    thread++)
    max_iter = max(max_iter, ccc_max_iter[thread]);
// if some thread has executed past iteration i,
// this iteration must be skipped
if(max_iter>i)
{
    record_skipped_iter(i, omp_get_thread_num());
    return TRUE;
}
}

// reset ccc_max_iter[] to refer to the iteration in
// which each thread took its respective checkpoint
ccc_max_iter[omp_get_thread_num()] =
    get_skipped_iters(omp_get_thread_num())[0];
#pragma omp flush(ccc_max_iter)

// We now know that all threads have begun their
// checkpoints. This thread may now finish its
// checkpoint.

// Since no thread may now be blocked on entry into an
// ordered region make C3's implementation of ordered
```

```
//    synchronization blocking again.
ccc_ordered_blocking();

return FALSE; // this iteration will not be skipped
}

// Performs the restore portion of a checkpoint, recreating
//    the call stack of the iteration this thread took its
//    checkpoint in, and replaying any iterations skipped by
//    this thread
void replay_loop_body() {
    // This thread must finish up the checkpoint by reentering
    //    the ordered construct inside/before which this
    //    thread took its checkpoint. Note that since this
    //    thread has skipped a number of iterations, it cannot
    //    use the OpenMP ordered construct for this purpose as
    //    it would not assure the correct execution order of
    //    iterations since it cannot take into account the
    //    skipped iterations. Therefore, the thread will
    //    reenter the ordered construct's body using C3's own
    //    implementation of ordered.

    // continue with the checkpoint, switch from CCC_REWIND
    // mode to CCC_REPLAY mode
    ccc_explicit_checkpoint();
```

```
// Start CCC_REPLAY'ing the execution context.
assert(ccc_mode == CCC_REPLAY);

// Determine the iteration number of the last iteration
// with which we will be using our own implementation
// of ordered (the maximum iteration number among the
// skipped iterations). By checking against this
// number we will know later on when we can resume
// using OpenMP's implementation of ordered.
max_iter_ccc_ordered=-1;

#pragma omp barrier

#pragma omp critical (maxIterOrdered_Section)
{
    if(i>max_iter_ccc_ordered)
        max_iter_ccc_ordered =
            get_max_skipped_iter(omp_get_thread_num());
}

// Set the stack cursor.
ccc_stack_cursor = saved_stack_top;

// The loop below executes the remainder of the iteration
// inside which this thread took its checkpoint as well
// as any iterations it skipped. Since its first
// skipped iteration is the one it took its checkpoint
```

```
// in, the value of i in the first iteration in the
// loop below will be the same as when this thread
// began its checkpoint. Since at the beginning of the
// loop below ccc_mode = CCC_REPLAY, the first
// iteration will reconstruct the call stack to its
// state at the time of the checkpoint and complete
// the checkpoint before finishing up that iteration of
// the main loop and continuing with the remaining
// skipped iterations.
// C3's implementation of the ordered synchronization will
// be used with all the skipped iterations
skipped_iterations =
    get_skipped_iters(omp_get_thread_num());
foreach(i in skipped_iterations)
{
    loop_body(i);
    // if a checkpoint was begun inside this iteration
    if(ccc_mode==CCC_REWIND)
    {
        // set the C3 implementation of ordered to not
        // block on entry into ordered regions
        ccc_ordered_nonblocking();

        // leave this function to begin skipping
        // iterations for the benefit of OpenMP's
```

```

        // implementation of ordered
        return;
    }
}

// function that encapsulates the body of the for loop that
// may contain an ordered region
void loop_body(int i) {
    // record the fact that this thread has reached iteration
    // i since the most recent checkpoint
    ccc_max_iter[omp_get_thread_num()] = i;
    #pragma omp flush (ccc_max_iter)

    if (ccc_mode == CCC_REPLAY)
        switch (*ccc_stack_cursor++) {
            ...
            case k: goto ccc_label_k;
            ...
        }
    ... loop body ...
    ...
    ccc_label_k:
    // if we have not yet reached the last skipped iteration,
    // use C3's implementation of ordered

```

```
if(i>=max_iter_ccc_ordered)
{
    // we use our own implementation of ordered
    ordered_body(C3_ORDERED, i);
}
// else, if this is the "border" iteration that is the
// first iteration for which we will be switching back
// to OpenMP's implementation of ordered
else if(i==max_iter_ccc_ordered)
{
    // synchronize using both the C3 implementation of
    // ordered and OpenMP's
    ordered_body(BOTH_ORDERED, i);
}
// else, if we are past the switching point, use OpenMP's
// implementation of ordered
else
    ordered_body(OMP_ORDERED, i);
}

// if the thread exited the ordered region because it
// began taking a checkpoint
if (ccc_mode == CCC_REWIND)
{
    // reset the set of iterations skipped by this thread
```

```
// since this thread is about to start skipping
// iterations from scratch
reset_skipped_iters(omp_get_thread_num());

// record that we are skipping the rest of this
// iteration (useful for when we will be getting
// back into executing this iteration when we are
// done taking the checkpoint)
record_skipped_iter(i, omp_get_thread_num());
// go to the end of the iteration and start skipping
goto end_loop_body;
}

// else, if no checkpoint was taken, finish executing this
// iteration
...
// In case no ordered region was executed during this
// iteration and this iteration used C3's
// implementation of ordered, inform the ordered region
// in iteration i+1 (if any) that it is allowed to
// proceed.
ccc_ordered_exit(i);

end_loop_body:
}
```

```
// function that encapsulates the body of the ordered region
// ordered_type - indicates whether ccc_ordered should use the
//   OpenMP implementation of ordered (=OMP_ORDERED), C3's
//   implementation (=C3_ORDERED) or both (=BOTH_ORDERED)
void ordered_body(int ordered_type, int i) {
    if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
        goto ccc_label;
}

// Ordered regions outside of parallel regions are not
// allowed.
assert(ccc_parallel_depth > 0)

// Record the height of the pc stack at the start of the
//   ordered region.
// If a checkpoint is taken inside this ordered region,
//   the pc stack pointer will need to be reset to this
//   height before the call stack may be recreated during
//   the CCC_REPLAY phase of the checkpoint.
int saved_stack_top = ccc_top_pc_stack();

ccc_label:
    // if we should use the OpenMP implementation of ordered
    if(ordered_type == OMP_ORDERED)
    {
```

```
        #pragma omp ordered
            ordered_app_body(i);
    }

    // else, if the thread should use C3's implementation of
    // ordered
    else if(ordered_type == C3_ORDERED)
    {
        ccc_ordered_enter(i);
        ordered_app_body(i);
        ccc_ordered_exit(i);
    }

    // else, if the thread should use both OpenMP's and C3's
    // implementations of ordered
    else if(ordered_type == BOTH_ORDERED)
    {
        ccc_ordered_enter(i);
        #pragma omp ordered
            ordered_app_body(i);
        ccc_ordered_exit(i);
    }

    // if this thread entered the ordered region normally and
    // neither took a checkpoint inside of it (if that were
    // true ccc_mode would be == CCC_REWIND), nor was
    // forced to take a checkpoint when it entered it (if
```

```

//    that were true then checkpoint_initiated would be >0)
if (ccc_mode == CCC_NORMAL && checkpoint_initiated==0) {
    ccc_pop_pc();
    // This thread is finished with this ordered region.
    if (!checkpoint_initiated) return;
// Else, if a checkpoint has begun (ccc_mode==CCC_REWIND or
// checkpoint_initiated>0)
} else {
    // ordered regions may not be nested inside other
    // ordered regions
    assert(inside_ordered == 0)
    // Since ordered regions may not be closely nested
    //    inside of critical regions, if we are outside of
    //    any ordered region, we must not be inside any
    //    critical region.
    // We may now begin skipping iterations until any
    //    thread that is waiting to enter an ordered region
    //    may be awoken. Thus, continue returning until we
    //    leave this iteration.
    return;
}
}

// function containing the transformed application code of the
// ordered region

```

```

void ordered_app_body(int i) {
    if (ccc_mode == CCC_RESTART || ccc_mode == CCC_REPLAY)
        goto ccc_label;

    // If checkpoint_initiated is true, then one of two
    // conditions holds,
    // - The thread entered the ordered region because another
    // thread exited the ordered region of the immediately
    // preceding iteration in order to force this thread to
    // checkpoint itself.
    // - The application is in the middle of a checkpoint and
    // this thread started its checkpoint inside of an
    // ordered region. The thread executed a series of
    // returns and iteration skips in order to release the
    // ordered region. Now it must CCC_REPLAY the creation
    // of the execution context.

    // if no checkpoint has been initiated or this thread is
    // recreating its call stack, allow it to continue on
    // into the body of the ordered region.
    if (!checkpoint_initiated || ccc_mode == CCC_REPLAY) {
        inside_ordered=TRUE;

        // Call the application code of the ordered region
ccc_label:
        ...ordered body (transformed with jump table, etc.)...
    }
}

```

```

// if a checkpoint has begun inside this ordered region
if (ccc_mode == CCC_REWIND) goto end_ordered;

    inside_ordered=FALSE;
}

// If checkpoint_initiated is true
else

    // this thread must force a checkpoint of itself
    ccc_explicit_checkpoint();

end_ordered:
}

```

### 3.8.4.2 Critical and Ordered

While OpenMP does not allow ordered regions to be nested inside critical regions or other ordered regions, critical regions may be nested inside of ordered regions (i.e. it is possible for a thread to enter an ordered region and then enter one or more critical regions). As such, the code given above for critical sections is not entirely correct since it does not take ordered regions into account. In particular, when a thread exits the outermost critical section during the `CCC_REWIND` portion of the checkpointing process, it may still be inside an ordered region. As such, the above code needs to be modified as follows. When a thread exits the outermost critical section it needs to check whether `inside_ordered=1`. If so, it needs to continue rewinding until it has exited this ordered region and must proceed with

the rest of the protocol described above for waking up threads that may be blocked on entry into an ordered region.

### 3.8.5 Miscellany

#### 3.8.5.1 Finer-grained coordination.

The protocol routines shown above use two variables for checkpoint coordination, `threads_awoken[]` and `checkpoint_initiated`. The former is used to coordinate checkpointing at barriers and the latter is used to coordinate checkpointing at all other constructs (locks, `critical` regions and `ordered` regions). This approach was chosen for this description of the protocol is because of its simplicity. We call it the “eager” checkpointing coordination because it forces threads to take a checkpoint whenever they try to acquire a resource and after some thread has decided to take a checkpoint (it does not matter whether the resource in question was released by a thread that is trying to take a checkpoint).

A finer-grained method of coordination for non-barrier resources is also possible. By this we mean that each critical section name, lock, etc. has an associated coordination flag. When that resource is released in order to allow other threads to make progress to checkpointing, the releasing thread sets the flag of each resource being released to indicate to any threads that have or will block on the resource to checkpoint immediately. These flags are reset after the checkpoint is taken and each released resource is reacquired. This approach is called “lazy” checkpointing coordination because threads are forced to take a checkpoint only when they try to acquire a resource that has been released by a thread that is currently trying to take a checkpoint (i.e. only force checkpoints if it is required for correctness).

After a checkpoint has been initiated, the eager coordination method tends to

result in threads taking checkpoints sooner than the lazy method. A lazy checkpointing mechanism can be made to behave like an eager checkpointing mechanism by inserting potential checkpoint locations into the application immediately before acquisitions of non-barrier resources. (i.e. `ccc_set_lock`, entry into a critical section, etc.) There are performance trade-offs that can be made between these two approaches. For some applications, an eager approach may give better performance because threads spend less time waiting for other threads to checkpoint. For other applications, a lazy approach may give better performance because threads are more likely to take checkpoints only at the developer specified locations. For still other applications, a hybrid approach may give the best performance.

The system used for the experiments reported in Section 3.9 uses a pure lazy mechanism.

### 3.8.5.2 Environment routines

OpenMP provides a number of routines, such as `omp_get_num_threads` and `omp_get_thread_num` that a thread can use to query its running environment.  $C^3$  ensures that these functions return the same values on restart by overloading them with  $C^3$  routines that keep track of the values of the relevant parameters and reset them on restart. Note that in order to have the same number of threads allocated by the modified application on restart, this number of threads must be available in the system. The current  $C^3$  implementation assumes this to be the case.

OpenMP also provides routines for changing the execution environment, such as `omp_set_num_threads`. These functions are also handled in  $C^3$  by overloading them with specialized  $C^3$  routines, recording the values of the relevant variables and resetting them on restart.

### 3.9 Implementation and Experiments

This section reports on the implementation of the mechanisms described above. The  $C^3$  implementation currently supports most of the OpenMP specification. The only important feature that has not yet been implemented are ordered sections.  $C^3$  provides trivial support for nested parallelism in that only a single thread may be created inside a nested parallel region.

Application-level checkpointing increases application running time in two different ways. Even if no checkpoints are taken, the instrumented code executes more instructions than the original application to perform book-keeping operations that keep track of local variables and OpenMP-specific state. Furthermore, if checkpoints are taken, writing the checkpoints to disk adds to the execution time of the program. Because different fault/migration environments will require different checkpointing frequencies, we measure these two overheads separately.

To measure the overheads introduced by  $C^3$  to a real application, we evaluated it using the SPLASH-2 [214] benchmark suite and the OpenMP versions [17] of the NAS Parallel Benchmarks [18]. None of the codes in these benchmarks ran for longer than 1 hour on our test systems, meaning that they were not themselves vulnerable to hardware failures. However, since these codes represent realistic computations,  $C^3$ 's performance with these codes should be indicative of its performance on real-world large applications. While SPLASH-2 and NAS benchmarks give us general information, they provide little insight about how  $C^3$ 's transformations affect the overhead of OpenMP constructs. We used the EPCC microbenchmark suite [183] for this purpose.

One of the major strengths of application-level checkpointing is that the instrumented code is as portable as the original code. To demonstrate the portability of

our approach we ran experiments on four different platforms:

- **Linux/Athlon:** 2-way 1.73GHz Athlon SMP with 1GB of RAM and SUSE 8.0 Linux with a 2.4.20 kernel. Intel C++ Compiler Version 7.1. Codes: SPLASH-2.
- **Linux/IA64:** 4-way 1.5Ghz Itanium with 2GB RAM and RedHat Enterprise Linux 4.0 with a version 2.6.9 kernel. Intel C++ Compiler V8.1 using the `-O3` optimization level. Codes: NAS, EPCC.
- **Tru64/Alpha:** 4-way Compaq Alphaserver ES45 (1Ghz EV68 processors, 4GB RAM) running Tru64 V5.1A. Compaq C Compiler V6.5, with `-O3 -fast` optimization. (Lemieux cluster at the Pittsburgh Supercomputing Center). Codes: SPLASH-2, NAS, EPCC.
- **Solaris/Sparc:** 4-way Sun 420R (v9 SPARCII processors, 4GB RAM) running Solaris 9. SunPro 9 Compiler with `-fast` optimization level. Codes: NAS, EPCC.

For the Linux/Athlon platform experiments were run using both processors. On the remaining platforms, experiments were run using 2 or 4 processors. In this thesis only the results of the 4-way experiments are reported since the 2-way experiments show similar behavior. The SPLASH-2 experiments are the average of 5 runs. The NAS and EPCC experiments are the average of multiple runs (number of runs in each experiment is listed with its table), with the upper and lower 10<sup>th</sup> percentile of results discarded. To minimize error from accessing a networked disk, checkpoints were recorded to the local disk of each node. The only exception was with the SPLASH-2 benchmarks on Tru64/Alpha, where system scratch space was

used (efficient and reliable checkpoint storage is an orthogonal research topic of that is not addressed in this paper).

### 3.9.1 Application Overheads - SPLASH-2

The SPLASH-2 benchmarks were used to measure the effects of  $C^3$  on the performance of real applications. These experiments were run on the Linux/Athlon and Tru64/Alpha platforms. The `cholesky` benchmark was omitted because it ran for only a few seconds, which was too short for accurate overhead measurement. Also omitted were `volrend` because of licensing issues with the tiff library, and `fmm` because we could not get even the unmodified benchmark to run on our platforms. Table 3.1 describes the locations in the other SPLASH-2 codes where we placed `potential_checkpoint` calls. Table 3.2 shows the input sizes used in our experiments on both platforms.

#### 3.9.1.1 Checkpoint-free Overheads

In this experiment, we measured the running times of (i) the original codes, and (ii) the instrumented codes without checkpointing. Times were measured using the Unix `time` command and the results are reported in Tables 3.3 and 3.4 for Linux/Athlon and Tru64/Alpha, respectively.

On Linux/Athlon the overhead introduced by  $C^3$  was within this noise margin (estimated at 2-3%). For two applications, `water-squared` and `water-spatial` on Linux/Athlon, the instrumented codes ran faster than the original, unmodified applications. Further experimentation showed that this unexpected improvement arose largely from the superior performance of our heap implementation compared to the native heap implementation on this system. We concluded that the overhead

Table 3.1: Characteristics of SPLASH-2 Benchmarks

benchmark	Checkpoint location
fft	in <code>FFT1D()</code> , before FFT on each column
radix	in <code>slave_sort()</code> , after each barrier
lu-c	at end of <code>lu()</code> outermost loop
barnes	in <code>SlaveStart()</code> after each time step
ocean-c	in <code>slave()</code> after every step
radiosity	in <code>process_tasks()</code> before every task
raytrace	in <code>RayTrace()</code> before every job bundle
water-nsquared	in <code>MDMAIN()</code> at the end of each time step
water-spatial	in <code>MDMAIN()</code> at the end of each time step

of  $C^3$  instrumentation code for the SPLASH-2 benchmarks on the Linux/Athlon platform is small, and that it is dominated by other effects such as the quality of the heap implementation.

The Alpha/Tru64 results show that except for `radix` and `ocean-c`, the overheads due to  $C^3$ 's transformations are either negligible or negative. The overheads in `radix` and `ocean-c` arise from two different problems.

The overhead in `radix` comes from some of the details of how  $C^3$  performs its transformations. The state-saving mechanism described in Section 3.6 computes addresses of all local and global variables, which may prevent the compiler from allocating these variables to a register. For `radix`, it appears that this inability to register-allocate certain variables leads to a noticeable loss of performance. In order to avoid this overhead in the future,  $C^3$  will transition to a different stack-

Table 3.2: Benchmark input sizes

Benchmark	Linux/Athlon	Tru64/Alpha
fft	$2^{24}$ data points	$2^{26}$ data points
lu-c	5000×5000 matrix	12000×12000 matrix
radix	100,000,000 keys, radix=512	300,000,000 keys, radix=512
barnes	16384 bodies, 15 steps	Did not run on this platform
ocean-c	514×514 ocean, 600 steps	1026×ocean, 600 steps
radiosity	Large Room	Large Room
raytrace	Car Model, 64MB RAM	Car Model, 1GB RAM
water-nsquared	4096 molecules, 60 steps	12167 molecules, 10 steps
water-spatial	4096 molecules, 60 steps	17576 molecules, 40 steps

saving mechanism that uses the `ucontext` family of functions to record and restore portions of the stack without taking the addresses of stack variables or using `gotos` (as described in Section 3.2 of [139]).

Our experiments showed that the overhead in `ocean-c` execution comes from our heap implementation (replacing our heap implementation with the native heap eliminated this overhead), which was not as optimized for Tru64 as it was for Linux.

### 3.9.1.2 Checkpoint and Restart Overhead

Tables 3.5 and 3.6 show the time overhead of taking a single checkpoint and performing a single restart on Linux/Athlon and Tru64/Alpha, respectively. These numbers can be used in formulas containing particular checkpointing frequencies and hardware failure probabilities to derive the overheads for a long-running application.

Table 3.3: SPLASH-2 Linux/Athlon Experiments

Benchmark	Uninstrumented run time	$C^3$ -instrumented run time 0 checkpoints taken	$C^3$ -instrumentation overhead
fft	20s	20s	0%
lu-c	110s	110s	0%
radix	30s	31s	3%
barnes	103s	106s	3%
ocean-c	162s	162s	0%
radiosity	8s	8s	0%
raytrace	32s	34s	6%
water-nsquared	260s	223s	-14%
water-spatial	156s	141s	-9%

The checkpoint time was computed using the formula:

$ChkptCost = ChkptTime_{1Chkpt} - ChkptTime_{0Chkpt}$ , where

- $ChkptTime_{1Chkpt}$  = running time of the transformed application where it takes a single checkpoint and
- $ChkptTime_{0Chkpt}$  = running time of the same transformed application without taking any checkpoints.

. Thus,  $ChkptCost$  includes not only the time to write the data to disk but also miscellaneous effects such the impact that taking a checkpoint has on future cache behavior.

Computing the cost of restarting uses the formula:

Table 3.4: SPLASH-2 Tru64/Alpha Experiments

Benchmark	Uninstrumented run time	$C^3$ -instrumented run time 0 checkpoints taken	$C^3$ -instrumentation overhead
fft	68s	67s	-2%
lu-c	719s	724s	1%
radix	61s	70s	15%
ocean-c	153s	183s	20%
radiosity	13s	12s	-9%
raytrace	20s	20.4s	2%
water-nsquared	136s	140s	3%
water-spatial	214s	218s	2%

$RestartCost = (AftChkpt + AftRestarting) - ChkptTime\_1Chkpt$ , where

- $AftChkpt$  is the time from the start of the application’s main computation region (the region of code timed by the benchmark itself) until it finished taking a checkpoint,
- $AftRestarting$  is the time from the beginning of the restart execution until the end of the application’s main computation region, and
- $ChkptTime\_1Chkpt$  is the time to execute the transformed application while taking a single checkpoint.

Because the checkpoint will likely be read from in-memory buffers rather than directly from disk, this measurement factors out most overheads peculiar to our system setup (such as disk bandwidth, etc.) and only measures the overhead of the

recovery in addition to the overheads of disk access, which are already measured in the single checkpoint experiments.

The results for Linux/Athlon show that the checkpoint and restart times are fairly low for most applications. Indeed, they are significant only for applications for which checkpoint sizes are very large (`fft` and `radix`). As mentioned before, these checkpoints were saved to local disk on the machine. If they were saved to a networked file system, we would expect the overheads to be larger.

The Tru64/Alpha experiments show that there exists a correlation between the sizes of the checkpoints and the amount of time it takes to perform the checkpoint. Because the Tru64/Alpha experiments featured larger checkpoint sizes and the checkpoint files were written to the system scratch space rather than to a local disk, the overheads observed are higher than for the Linux/Athlon experiments.

The only code with a high restart overhead on Tru64/Alpha is `water-nsquared`, and it highlights an inefficiency in the  $C^3$  implementation used for these experiments. Note that `water-nsquared` takes 3.5 seconds to record a 16MB checkpoint but takes 388 seconds to recover. The reason for this is that `water-nsquared malloc()-s` a large number of individual objects: 194K. This is in contrast to the 18K objects that `water-spatial` allocates or the 65K allocated by `water-nsquared` given the input parameters used on Linux.  $C^3$ 's checkpointing code is optimized to use buffering when writing these objects to a checkpoint, but its restart code does not have such optimizations, so it performs one file read for every one of these objects. The cost of that many file reads, even to buffered files is very high and results in a long recovery time. By comparison if `water-nsquared` were run on Alpha using the Linux parameters, it would have a 70s restart overhead. The version of  $C^3$  used in the NAS and EPCC experiments includes an optimized mechanism

for reading the checkpoint files that eliminate this inefficiency.

`Ocean-c`'s restart overhead was measured to be negative. Since this negative overhead is within the variability of the timing results in this experiment, it appears to be an artifact of the fluctuations inherent to a networked file system.

Table 3.5: Overhead of Checkpoint and Restart on Linux/Athlon.

Benchmark	Checkpoint Size (MB)	Seconds per Checkpoint	Seconds per Restart
fft	765	43	22
lu-c	191	2	5
radix	768	43	24
barnes	569	4	10
ocean-c	56	1	4
radiosity	32	0	1
raytrace	68	0	2
water-nsquared	4	1	0
water-spatial	3	0	0

### 3.9.1.3 Checkpoint Size Comparison to SLC

This section compares  $C^3$  to system level checkpointing techniques. Since SLC solutions do not modify the source code of the application and in general do very little outside of checkpointing the application, their no-checkpoint overhead is generally close to 0%. Thus, the only remaining comparison to SLC solutions is in terms of per-checkpoint cost. Since the cost of a checkpoint is dominated by its size, we focus our comparison on the sizes of the checkpoints produced for different

Table 3.6: Overhead of Checkpoint and Restart on Tru64/Alpha.

Benchmark	Checkpoint Size (MB)	Seconds per Checkpoint	Seconds per Restart
fft	3074	363	32
lu-c	1103	136	7
radix	2294	285	36
ocean-c	224	68	*
radiosity	43	8	1
raytrace	1033	137	7
water-nsquared	16	3.75	388
water-spatial	12	3.5	17

applications by  $C^3$  and an SLC solution.

Since we are not aware of any SLC solutions for Tru64 that support multi-threaded programs, we focus on Linux/Athlon and compare  $C^3$  to BLCR [75], a checkpointer for applications running on Linux that use pthreads. Since the Intel OpenMP compiler is based on pthreads, BLCR could be used to checkpoint executables created by this compiler. The checkpoint sizes produced by both systems on the SPLASH-2 benchmarks are shown in Table 3.7.

For most of the codes, the difference between the checkpoint sizes of the two systems is minimal. The codes for which this is not true, **radix** and **barnes**, benefit from a feature of BLCR that is not found in  $C^3$ . Since BLCR is designed to work only with Linux and operates at the kernel-level, it optimizes state-saving by checkpointing only those pages that were actually allocated to the application by the kernel. On Linux, calls to `malloc` do not allocate any physical pages but

merely reserve address space. A physical page is allocated for a logical page only when that logical page is written to for the first time. In contrast,  $C^3$  works above the Linux kernel and therefore cannot keep track of this information. As such, it saves all the pages that have been `malloc`-ed by the application.

This difference between  $C^3$  and BLCR is very relevant to the SPLASH-2 benchmarks because most memory in these codes is allocated near the beginning of execution and then written to over the rest of the execution. In particular, `barnes` and `radix` allocate 130MB and 765MB of memory respectively at the start of execution, but do not write to all of this memory until they near the end of their execution. As such, for these benchmarks the size of the checkpoint taken by BLCR is much smaller at the beginning of their executions than at the end. This difference is displayed in Table 3.7 by showing the full range of checkpoint sizes recorded by BLCR for `barnes` and `radix`.

Table 3.7: SPLASH-2 Linux/Athlon  $C^3$  vs BLCR Checkpoint Sizes

Benchmark	$C^3$ Checkpoint Size (MB)	BLCR Checkpoint Size (MB)
fft	765	770
lu-c	191	192
radix	768	284-764
barnes	569	55-130
ocean-c	56	52
radiosity	32	29
raytrace	68	33
water-nsquared	4	5
water-spatial	3	3

This experiment shows that the baseline checkpoint sizes for  $C^3$  and BLCR are quite similar. While  $C^3$  suffers from a lack of kernel-level knowledge, it would be possible to bring  $C^3$ 's checkpoint sizes fully in line with BLCR's if it were augmented with system-specific modules that can keep track of which pages have been written to. While this would make  $C^3$  less portable, the new non-portable component would be relatively small and may be worth-while for the sake of performance. While checkpoint sizes can further be improved via techniques such as incremental checkpointing [166] and compression [170], these techniques can be used by both system-level and application level-solutions. Furthermore, by working at the application-level,  $C^3$  can reduce its checkpoint sizes even more by using compiler techniques such as [168] and [133], which are not available at the system-level.

### 3.9.2 Application Overheads - NAS

We used the NAS benchmarks to determine the effects that  $C^3$  has on an application's performance. These experiments were performed using a more recent version of  $C^3$  that incorporated efficiency optimizations and supported a much broader subset of the OpenMP specification.

#### 3.9.2.1 Checkpoint-free Overheads

Our first experiment measures the overhead of  $C^3$ 's instrumentation of the application code. This requires the measurement of the running times of (i) the original codes, and (ii) the instrumented codes without checkpointing. Running times were taken from the output of each benchmark as this gives us the running time of the section of the code performing the computational behavior represented by that benchmark. Results for the Linux/IA64, Tru64/Alpha and Solaris/Sparc runs are

shown in Table 3.8, Table 3.9 and Table 3.10, respectively. We show results for all benchmark/problem size combinations that we could run on each platform and for each combination list the running time of the original application and the percent overhead resulting from transforming the application so that it can checkpoint itself, without taking any checkpoints. Results for MG on Alpha/Tru64 are missing because this code fails to run on this platform.

The tables show that for most codes, the overhead introduced by  $C^3$  was generally small. There are a few important things that are apparent.

- In general, larger input sizes lead to smaller overheads. This happens because  $C^3$ 's transformations and bookkeeping may make OpenMP constructs more expensive. When running on larger inputs, applications typically spend less time on OpenMP constructs and more time doing useful work, reducing  $C^3$ 's overhead. We believe this to be the typical real world behavior since HPC applications typically feature large inputs. In two cases (IS on Linux/IA64 and MG on Solaris/Sparc) the overheads spike between problem size S and W but this is followed by a significant drop for the larger problem sizes. Except for MG on Linux/IA64, the overhead is  $< 4\%$  for all benchmarks on all platforms and  $< 1\%$  in almost every case.
- In some cases overheads are negative. In almost every case they are tiny relative to the running time of the application and can be regarded as an expected variance due to modifications to the program's source code, which can be either positive or negative. The only exception is IS on Linux/IA64 with problem size B, where the no-checkpoint overhead is  $-25\%$ . This happens because our transformations may take addresses of stack variables. In particular, by taking the address of array `prv_buff1` in function `rank` we

cause the Intel compiler to perform some more aggressive optimizations that are otherwise not performed. This effect is not seen under -O2 optimization level.

- MG on Linux/IA64 is the only case where the 0-checkpoint overhead is greater than a few % on all problem sizes. This is because  $C^3$  creates aliases by taking addresses of variables that it wishes to checkpoint, which can confuse the compiler and prevent certain optimizations. While the version of  $C^3$  used for these experiments was more advanced than the version used for SPLASH-2 and performed compiler analyses to avoid the need to take the addresses of most variables, this is always possible and in MG on Linux/IA64 it resulted in high overheads. This overhead can be eliminated by using the `ucontext` family of functions to save stack state.
- One important lesson can be learned by looking at where the overheads do *not* come from.  $C^3$  places a large number of statement labels and `gotos` throughout the program to enable it to Replay the creation of the stack. Since these `gotos` complicate the application's control flow, one could imagine them confusing the compiler and causing high overheads. It turns out that this aspect of  $C^3$ 's transformations have little effect on performance (Note that the switch to `getcontext/setcontext`-based stack management [139] will eliminate these `gotos` as well).

Table 3.8: NAS Linux/IA64 Original Runtimes, No-Checkpoint Overheads and Checkpoint Sizes (40 runs each)

Benchmark	S		W		A	
BT	0.12s	1.60%	4.53s	0.08%	199s	-0.22%
CG	0.052s	16.47%	0.22s	7.31%	1.88s	3.40%
EP	0.981s	-1.46%	1.95s	-1.51%	15.6s	-1.57%
FT	0.143s	7.07%	0.29s	5.91%	5.99s	5.04%
IS	0.00079s	40.73%	0.022s	192.50%	0.4s	0.41%
LU	0.045s	0.68%	4.6s	1.28%	42.7s	1.37%
MG	0.0073s	36.76%	0.42s	41.83%	3.4s	24.35%
SP	0.326s	0.66%	16.7s	0.79%	130s	1.15%
Benchmark	B		C			
BT	830s	0.50%				
CG	77s	1.34%				
EP	62s	-1.65%	250s	-1.84%		
FT						
IS	9.1s	-24.99%				
LU	248s	0.89%				
MG	16s	25.99%				
SP	523s	0.96%				

Table 3.9: NAS Tru64/Alpha Original Runtimes, No-Checkpoint Overheads and Checkpoint Sizes(35 runs each)

Benchmark	S		W		A	
BT	0.24s	1.05%	7.8s	1.57%	227s	0.67%
CG	0.096s	17.78%	0.71s	1.27%	3.2s	-3.52%
EP	1.3s	-0.52%	2.7s	-0.28%	21s	-0.25%
FT	0.19s	-1.46%	0.42s	-0.10%	8s	-0.24%
IS	0.0017s	13.13%	0.071s	2.38%	1.2s	0.71%
LU	0.056s	6.08%	7.4s	1.34%	105s	4.51%
SP	0.35s	7.17%	23s	2.49%	137s	10.36%
Benchmark	B		C			
BT	977s	0.69%				
CG	173s	-0.05%	912s	-2.30%		
EP	85s	-0.24%	338s	-0.20%		
FT	96s	0.30%				
IS	11s	0.03%	70s	0.03%		
LU	325s	0.03%	1702s	-0.34%		
SP	623s	4.87%	2499s	3.57%		

Table 3.10: NAS Solaris/Sparc Original Runtimes, No-Checkpoint Overheads and Checkpoint Sizes (40 runs each)

Benchmark	S		W		A	
BT	0.62s	4.23%	24.8s	3.43%	849s	-0.10%
CG	0.16s	3.40%	1.23s	-1.17%	9.7s	2.57%
EP	2.65s	0.90%	5.28s	0.80%	42s	1.11%
FT	0.72s	-1.54%	1.54s	1.36%	29s	0.37%
IS	0.0068s	3.52%	0.29s	4.30%		
LU	0.19s	0.58%	31.8s	6.03%	248s	2.49%
MG	0.018s	5.92%	1.15s	46.57%	14.4s	-0.03%
SP	0.79s	7.72%	72.8s	4.25%	608s	1.34%
Benchmark	B		C			
BT						
CG	699s	-0.55%				
EP	168s	0.93%	670s	0.43%		
FT						
IS						
LU	1958s	0.82%				
MG	69s	-1.16%				
SP						

### 3.9.2.2 Cost of Checkpointing

Tables 3.11, 3.12 and 3.13 show the amount of time it takes to record a single checkpoint on Linux/IA64, Tru64/Alpha and Solaris/Sparc respectively. For each application/platform, we show for the largest problem size that we could run, the absolute time to take a checkpoint (in seconds), the relative time as a percentage of the original application's running time and the size of the checkpoint. The checkpoint time was computed using the formula used for the SPLASH-2 experiments. For all the applications, it does not matter when during the application's execution the checkpoint is taken. The one exception, LU on Tru64/Alpha, is discussed below.

The cost of checkpointing is generally low, on the order of a few seconds, rising beyond a few seconds only for checkpoints larger than 1GB. In cases where the original application runs for a very short time relative to the amount of state it uses, even these few seconds are large relative to the total program runtime. This emphasizes the need for compiler analyses to reduce checkpoint sizes.

In some cases the time to record a checkpoint is negative. This is generally a small fraction of the running time of the application itself and is an expected runtime variance due the non-trivial perturbation of the system caused by a checkpoint. However, for LU on Tru64/Alpha the time to record one checkpoint is consistently negative to a large degree. This effect is strongest for checkpoints taken at the start of LU's execution and drops to 0s for those taken towards the end. The source of this phenomenon is that the global array `u` is touched during checkpointing, which causes cache hit rates to improve. It is not clear which details of Lemieux's hardware and system setup contribute to this effect but a similar effect is seen on restart with LU and CG.

Table 3.11: NAS Linux/IA64 1-Checkpoint Times in Secs and % of Runtime(35 runs each)

Benchmark	S			W			A		
BT	3.2E-05s	0.03%	2.7MB	0.074s	1.63%	17.6MB	1.2s	0.60%	306MB
CG	0.012s	24.00%	3.1MB	0.053s	23.54%	17.8MB	0.15s	8.15%	60MB
EP	0.01s	1.24%	4.4MB	0.011s	0.55%	4.3MB	0.016s	0.10%	4.3MB
FT	0.035s	24.71%	13.5MB	0.07s	23.66%	26.9MB	1.1s	18.87%	419MB
IS	0.0053s	668.74%	1.1MB	0.037s	167.11%	12.3MB	0.25s	63.46%	96MB
LU	-5.1-05s	-0.11%	0.71MB	0.023s	0.49%	6.7MB	0.023s	0.05%	45MB
MG	0.0075s	102.75%	1.5MB	0.030s	7.16%	8.1MB	1.8s	53.74%	438MB
SP	0.0062s	1.90%	0.94MB	0.073s	0.44%	15MB	0.52s	0.40%	80MB
Benchmark	B			C					
BT	17s	2.02%	1.2GB						
CG	1.6s	2.07%	428MB						
EP	0.012s	0.02%	4.3MB	-0.003s	0.00%	4.3MB			
FT									
IS	1.3s	14.02%	384MB						
LU	-1.3s	-0.51%	174MB						
MG	2s	12.66%	438MB						
SP	1.1s	0.21%	317MB						

Table 3.12: NAS Tru64/Alpha 1-Checkpoint Times in Secs and % of Runtime(40 runs each)

Benchmark	S			W			A		
BT	-0.0005s	-0.22%	2.7MB	0.090s	1.16%	17.6MB	0.94s	0.41%	307MB
CG	0.013s	13.95%	3.1MB	0.079s	11.14%	17.8MB	0.30s	9.28%	60MB
EP	0.011s	0.85%	4.3MB	0.016s	0.59%	4.3MB	0.009s	0.04%	4.3MB
FT	0.036s	19.15%	13.5MB	0.092s	22.24%	27MB	1.85s	23.17%	419MB
IS	0.0085s	491.82%	1.0MB	0.079s	110.94%	12.3MB	0.57s	45.88%	96MB
LU	-0.0004s	-0.73%	0.7MB	-0.0096s	-0.13%	6.7MB	-0.86s	-0.84%	45MB
SP	0.0083s	2.36%	0.93MB	0.076s	0.33%	15MB	0.42s	0.31%	80MB
Benchmark	B			C					
BT	0.77s	0.08%	1.2GB						
CG	1.7s	0.96%	428MB	18.7s	2.05%	1.1GB			
EP	0.014s	0.02%	4.3MB	-0.03s	-0.01%	4.3MB			
FT	6.1s	6.35%	1.7GB						
IS	2.5s	23.73%	384MB	2.9s	4.11%	1.5GB			
LU	-5.1s	-1.56%	174MB	-40s	-2.32%	678MB			
SP	1.1s	0.18%	317MB	7.1s	0.28%	1.3GB			

Table 3.13: NAS Solaris/Sparc 1-Checkpoint Times in Secs and % of Runtime(40 runs each)

Benchmark	S			W			A		
BT	0.0043s	0.70%	2.6MB	0.45s	1.80%	17MB	4.4s	0.52%	306MB
CG	0.034s	20.95%	3.1MB	0.16s	13.29%	18MB	0.38s	3.94%	60MB
EP	0.078s	2.94%	4.2MB	0.066s	1.26%	4.2MB	-0.063s	-0.15%	4.2MB
FT	0.14s	19.05%	13.4MB	0.24s	15.87%	27MB	4.3s	14.78%	419MB
IS	0.017s	256.06%	0.98MB	0.024s	8.40%	12MB			
LU	-0.00051s	-0.26%	0.64MB	0.24s	0.77%	6.6MB	1.1s	0.44%	45MB
MG	0.031s	170.05%	1.4MB	0.16s	14.10%	8MB	0.7s	4.84%	438MB
SP	0.01s	1.30%	0.87MB	1.09s	1.49%	15MB	1.3s	0.21%	80MB
Benchmark	B			C					
BT									
CG	1.7s	0.25%	428MB						
EP	-0.4s	-0.24%	4.2MB	0.75s	0.11%	4.2MB			
FT									
IS									
LU	3.7s	0.19%	174MB						
MG	6.6s	9.57%	438MB						
SP									

### 3.9.2.3 Cost of Restarting

Computing the cost of restarting uses the formula (note that it is slightly different from the formula used in the SPLASH-2 experiments):

$RestartCost = (BefChkpt + AftRestarting) - ChkptTime\_0Chkpt$ , where

- *BefChkpt* is the time from the start of the application's main computation region (the region of code timed by the benchmark itself) until it began to take a checkpoint,
- *AftRestarting* is the time from the beginning of the restart execution until the end of the application's main computation region, and
- *ChkptTime\_0Chkpt* is the time to execute the transformed application without taking any checkpoints.

Table 3.14, Table 3.15 and Table 3.16 show the cost of restarting on Linux/IA64, Tru64/Alpha and Solaris/Sparc, respectively. In each case we show the absolute restart cost in seconds and the relative restart time as a percentage of the original application's running time.

For Tru64/Alpha, Linux/IA64 and Solaris/SPARC the restart overheads are consistently low. The only anomaly is the very negative restart times for CG and LU Tru64/Alpha, apparently caused by the restart process touching key arrays and causing the cache hit rates to improve. This appears to be a common effect on this platform, affecting both checkpoint and restart results of different NAS applications as the system underwent updates during the course of our experiments.

Table 3.14: NAS Linux/IA64 Restart Times in Secs and % of Runtime(35 runs each)

Benchmark	S		W		A	
BT	-8.6E-05s	-0.07%	0.067s	1.48%	0.41s	0.21%
CG	0.012s	22.96%	0.028s	12.70%	0.14s	7.60%
EP	0.018s	1.87%	0.0094s	0.48%	0.0082s	0.05%
FT	0.022s	15.21%	0.036s	12.08%	0.68s	11.30%
IS	0.0096s	1201.22%	0.026s	119.14%	0.14s	35.50%
LU	-0.00017s	-0.39%	0.028s	0.62%	0.035s	0.08%
MG	0.0092s	126.94%	0.011s	2.67%	0.087s	2.54%
SP	0.013s	3.90%	0.073s	0.44%	0.24s	0.19%
Benchmark	B		C		C	
BT	7.3s	0.87%				
CG	0.45s	0.59%				
EP	-0.0083s	-0.01%	1.2s	0.50%		
FT						
IS	0.72s	7.83%				
LU	-11s	-4.56%				
MG	0.046s	0.28%				
SP	0.15s	0.03%				

Table 3.15: NAS Tru64/Alpha Restart Times in Secs and % of Runtime(40 runs each)

Benchmark	S		W		A	
BT	-0.0027s	-1.12%	-0.053s	-0.68%	-6.2s	-2.71%
CG	0.049s	50.78%	0.20s	27.81%	0.32s	10.03%
EP	0.53s	39.57%	0.69s	25.74%	2.7s	12.83%
FT	0.44s	234.87%	0.69s	164.97%	2.7s	33.76%
IS	0.023s	1333.03%	0.22s	315.60%	0.59s	47.42%
LU	-0.0018s	-3.30%	-0.058s	-0.78%	-8.1s	-7.88%
SP	0.50s	141.13%	2.6s	11.30%	12.6s	9.19%
Benchmark	B		C			
BT						
CG	10.4s	6.00%	-91s	-10.00%		
EP	9.8s	11.62%	41s	12.03%		
FT	21s	21.97%				
IS	2s	19.57%	9s	12.94%		
LU	-32s	-9.97%	-240s	-14.08%		
SP	62s	10.00%	183s	7.33%		

Table 3.16: NAS Solaris/Sparc Restart Times in Secs and % of Runtime(40 runs each)

Benchmark	S		W		A	
BT	0.0081s	1.31%	0.41s	1.65%	6.6s	0.77%
CG	0.049s	30.04%	0.21s	16.61%	0.53s	5.47%
EP	0.047s	1.77%	0.014s	0.27%	-0.21s	-0.49%
FT	0.16s	22.16%	0.28s	18.09%	4.2s	14.53%
IS	0.025s	364.98%	0.16s	54.72%		
LU	-0.00048s	-0.24%	0.1s	0.33%	1.9s	0.77%
MG	0.021s	114.01%	0.04s	3.48%	-0.12s	-0.84%
SP	0.026s	3.23%	0.76s	1.04%	0.67s	0.11%
Benchmark	B		C			
BT	21s	0.55%				
CG	4.5s	0.64%				
EP	-0.82s	-0.49%	9s	1.34%		
FT						
IS						
LU	10s	0.53%				
MG	0.2s	0.29%				
SP						

### 3.9.2.4 Checkpoint Size Comparison to SLC

This section compares the sizes of checkpoints generated by  $C^3$  to those generated by SLC techniques on the NAS benchmarks. Since we are not aware of any SLC solutions for Tru64 or Solaris that support multi-threaded applications, this section again focuses on Linux and the BLCR system-level checkpointer. Because BLCR cannot currently checkpoint IA64 applications and only has experimental support for 2.6.\* Linux kernels, this experiment was performed on the Linux/Athlon platform used in the SPLASH-2 experiments.

The checkpoint sizes generated by  $C^3$  and BLCR are shown in Table 3.17. In almost all cases the two systems produce checkpoints of the same size. The only exception is for FT, where BLCR produces checkpoints of different sizes, depending on whether the checkpoint was taken at the beginning of FT's execution (smaller checkpoint) or the end (larger checkpoint). This is the same as the effect seen in the SPLASH-2 experiments above and is caused by the fact that BLCR only checkpoints the pages in memory that have been written to since the start of the application.

These experimental results further underscore the lessons learned from the SPLASH-2 checkpoint size experiments above and similar comparisons between the sequential version of  $C^3$  and Condor in [55] and [189]. The checkpoint sizes generated by  $C^3$  are quite competitive to those generated by system-level checkpointers.  $C^3$ 's checkpoint sizes are generally equal to or smaller than those generated by Condor. They are equal to or sometimes larger than those generated by BLCR. The cases where  $C^3$  loses are due to the fact that it lacks some low-level information available to system-specific checkpointers. However, it can easily be extended via small system-specific modules to take advantage of this information,

bringing its checkpoint sizes in line with BLCR and other SLC systems that may use such information. Importantly, unlike its system-level counterparts,  $C^3$  can further optimize its checkpoints via the use compiler analyses such as [168] and [133].

Table 3.17: NAS Linux/Athlon  $C^3$  vs BLCR Checkpoint Sizes

Benchmark	S		W		A	
	$C^3$	BLCR	$C^3$	BLCR	$C^3$	BLCR
BT	2.6MB	2.5MB	17MB	17MB	307MB	298MB
CG	3.0MB	2.9MB	18MB	17MB	60MB	56MB
EP	2.2MB	2.4MB	2.2MB	2.4MB	2.2MB	2.4MB
FT	13.4MB	9.6MB-13.6MB	27MB	11MB-27MB	419MB	32MB-420 MB
IS	0.95MB	0.40MB	12MB	2.1MB - 9.1MB		
LU	0.62MB	0.79MB	6.6MB	6.6MB	45MB	44MB
MG	1.3MB	1.43MB	7.7MB	7.9MB	435MB	435MB
SP	0.84MB	1.00MB	15MB	14MB		
Benchmark	B		C			
BT						
CG	428MB	396MB				
EP	2.2MB	2.4MB	2.2MB	2.4MB		
FT						
IS						
LU						
MG	435MB	435MB				
SP						

### 3.9.3 Detailed Examination of Overheads

This section uses the EPCC microbenchmarks (divided into Synchronization, Scheduling and Array) to examine in detail the overheads of  $C^3$ 's transformations on individual OpenMP constructs. It uses the same version of  $C^3$  as the NAS experiments from Section 3.9.2. We report the times recorded for three configurations:

- *Original*: the original microbenchmarks.

- *C<sup>3</sup>NoPChkpt*: the microbenchmarks after transformation by *C<sup>3</sup>* but with no potential checkpoint locations inside the code. In realistic applications most OpenMP directives will not contain a potential checkpoint location.
- *C<sup>3</sup>PChkpt*: the microbenchmarks that have been modified to contain a potential checkpoint location inside each OpenMP construct being tested.

All times are in microseconds.

### 3.9.3.1 Synchronization

The results of the Synchronization benchmarks for Linux/IA64, Tru64/Alpha and Solaris/Sparc are shown in Table 3.18, Table 3.19 and Table 3.20, respectively. Below are the description of each microbenchmark and the lessons learned from it.

- **Parallel**: the cost of entering and exiting a parallel region. Since *C<sup>3</sup>* does not transform parallel regions with no potential checkpoint locations, they have the same cost under *Original* and *C<sup>3</sup>NoPChkpt*. A parallel region that contains a potential checkpoint is significantly more expensive than one that does not because *C<sup>3</sup>* uses a number of global `threadprivate` variables with `copyin` to maintain accounting state (the overhead comes primarily from the `copyin`). While parallel regions are much more expensive under *C<sup>3</sup>PChkpt*, they still only take up tens to a few hundred microseconds. Since most applications open few parallel regions, this overhead is unlikely to have any effect on realistic applications.
- **For**: the cost of a parallel for loop with a single iteration per thread. Because the `for` loop in this benchmark was followed by an implicit barrier, in

*C<sup>3</sup>NoPChkpt* and *C<sup>3</sup>PChkpt* the implicit barrier is removed and it is followed by an explicit call to *ccc\_barrier*. Since on all platforms *Original* and *C<sup>3</sup>NoPChkpt* have almost identical cost, it appears that this barrier transformation has little effect. Under *C<sup>3</sup>PChkpt* the cost of **for** is significantly larger because in addition to the barrier transformation 1. it has to record that a given **for** loop iteration has been allocated to a given thread and 2. as part of our Reimplementation of private variables, *C<sup>3</sup>* extracts the bodies of parallel **for** loops and turns them into functions. Any local variables used inside the body of the **for** are passed into the function as arguments.

- **Parallel For:** the cost of a **parallel for** directive, which is a **parallel** directive that contains a single parallel **for** loop with a single iteration per thread. *C<sup>3</sup>* breaks up **parallel for** directives into a normal form, which is a **parallel** directive that contains a **for** directive. On Linux/IA64 the *C<sup>3</sup>NoPChkpt* version's **parallel for** costs twice as much as the native version, while on Solaris/Sparc it is 32% more and 0% on Tru64/Alpha. The cost for *C<sup>3</sup>PChkpt* is approximately the sum of the cost of **parallel** and **for** directives.
- **Barrier:** the cost of a single barrier. *Original* uses the native compiler's implementation of barrier while *C<sup>3</sup>NoPChkpt* and *C<sup>3</sup>PChkpt* use the *C<sup>3</sup>* implementation. The cost of both implementations is similar on all platforms, with a maximum overhead of 37% on Solaris/Sparc. This is a minor overhead since the primary cost of a barrier is the time lost to threads waiting for each other to reach the barrier rather than the cost of the barrier itself. If this proves to be an important overhead for some applications, it would be trivial

to create multiple implementations of barrier and use the fastest one for each given platform.

- **Single:** the cost of a `single` directive, which allocates a piece of code for execution by some thread, followed by an implicit barrier. This directive has the same cost in *Original* and *C<sup>3</sup>NoPChkpt* (despite the same barrier transformation as with `for`) but is higher in *C<sup>3</sup>PChkpt* (by 3  $\mu$ s - 14  $\mu$ s) because of the additional accounting and privatization code inserted by *C<sup>3</sup>* in this version.
- **Critical:** the cost of a `critical` section. Again, *Original* and *C<sup>3</sup>NoPChkpt* have critical sections of the same cost. On Linux/x86 and Tru64/Alpha critical sections are more expensive in *C<sup>3</sup>PChkpt* because of the additional accounting code required to deal with potential checkpoint locations inside them. However, on Solaris the cost of a critical section under *C<sup>3</sup>PChkpt* is very negative. The reason is the use of `setjmp` to exit critical sections at checkpoint-time. On this platform a call to `setjmp` in a critical section speeds up the execution of the delay loop the EPCC microbenchmarks put inside OpenMP constructs. Since `setjmp` is such a low-level function it is difficult to determine the source of this effect.
- **Lock/Unlock:** the cost of acquiring and releasing a lock. It is higher in *C<sup>3</sup>NoPChkpt* and *C<sup>3</sup>PChkpt* than in *Original* because of the additional testing performed by *C<sup>3</sup>* lock functions to determine whether a checkpoint has begun.
- **Atomic:** the cost of `atomic` directives. It is the same in all configurations because *C<sup>3</sup>* pulls the right hand side of the atomic update expression outside

the `atomic` directive, as discussed in Section 3.7.2.1.

- **Reduction:** The additional cost of a parallel region with a reduction variable computed at the end of the region over that of a regular parallel region. The cost of reduction is very similar for *Original* and *C<sup>3</sup>NoPChkpt*. For *C<sup>3</sup>PChkpt* we Reimplement OpenMP's reduction functionality, resulting in a few  $\mu s$  overhead on Tru64/Alpha and Solaris/Sparc and a few  $\mu s$  reduction in cost on Linux/IA64.

Table 3.18: Linux/IA64 Synchronization Microbenchmarks (in  $\mu sec$ ) (60 runs each)

<i>Type</i>	<i>Parallel</i>	<i>For</i>	<i>Parallel For</i>	<i>Barrier</i>	<i>Single</i>	<i>Critical</i>
<i>Original</i>	3.32	1.9	3.4	1.88	2.08	0.47
<i>C<sup>3</sup> No Checkpoint</i>	3.36	2.1	6.1	1.97	1.9	0.47
<i>C<sup>3</sup> Checkpoint</i>	33	28	65	1.98	5.4	0.99
<i>Type</i>	<i>Lock/Unlock</i>	<i>Atomic</i>	<i>Reduction</i>			
<i>Original</i>	0.47	2.47	2.57			
<i>C<sup>3</sup> No Checkpoint</i>	0.69	2.41	2.87			
<i>C<sup>3</sup> Checkpoint</i>	0.69	2.33	1.44			

Table 3.19: Tru64/Alpha Synchronization Microbenchmarks (in  $\mu sec$ ) (90 runs each)

<i>Type</i>	<i>Parallel</i>	<i>For</i>	<i>Parallel For</i>	<i>Barrier</i>	<i>Single</i>	<i>Critical</i>
<i>Original</i>	4.5	1.69	6.46	1.72	1.2	0.98
<i>C<sup>3</sup> No Checkpoint</i>	4.53	2.28	6.33	2.05	3.13	0.99
<i>C<sup>3</sup> Checkpoint</i>	150	111	211	2.06	6.39	4.2
<i>Type</i>	<i>Lock/Unlock</i>	<i>Atomic</i>	<i>Reduction</i>			
<i>Original</i>	2.95	0.17	1.47			
<i>C<sup>3</sup> No Checkpoint</i>	3.91	0.17	1.45			
<i>C<sup>3</sup> Checkpoint</i>	3.87	0.17	3.82			

Table 3.20: Solaris/Sparc Synchronization Microbenchmarks (in  $\mu\text{sec}$ ) (60 runs each)

<i>Type</i>	<i>Parallel</i>	<i>For</i>	<i>Parallel For</i>	<i>Barrier</i>	<i>Single</i>	<i>Critical</i>
<i>Original</i>	8.1	3.5	9.8	1.9	1.3	0.55
<i>C<sup>3</sup> No Checkpoint</i>	8.3	4.2	13	2.6	2.1	0.56
<i>C<sup>3</sup> Checkpoint</i>	89	72	161	2.8	16	-2.2
<i>Type</i>	<i>Lock/Unlock</i>	<i>Atomic</i>	<i>Reduction</i>			
<i>Original</i>	0.49	0.89	2.1			
<i>C<sup>3</sup> No Checkpoint</i>	1.0	0.89	2.2			
<i>C<sup>3</sup> Checkpoint</i>	1.0	0.92	9.5			

### 3.9.3.2 Scheduling

The Scheduling benchmarks measure the cost of using different parallel `for` loop iteration scheduling policies (i.e. which thread will get which loop iteration). The benchmark runs 128 iterations of the `for` loop on each thread and we report the cost on a per-iteration basis (i.e. total cost / 128). Since this benchmark's `for` loops are followed by an implicit barrier,  $C^3$  performs the barrier transform described above.

The results for the Scheduling benchmarks for Linux/IA64, Tru64/Alpha and Solaris/Sparc are shown in Table 3.21, Table 3.22 and Table 3.23, respectively. For static scheduling it is possible to not specify a chunk size.

- **Static:** for a given chunk size `cksz`, iterations are allocated in deterministic round-robin order in units of `cksz` iterations.  $C^3$ 's barrier transformation causes little difference in the per-iteration cost under *Original* and *C<sup>3</sup>NoPChkpt* on Linux/IA64 and Tru64/Alpha and a noticeable but small (in absolute terms) difference on Solaris/Sparc. *C<sup>3</sup>PChkpt* has a larger cost because  $C^3$  performs additional accounting work to record the which iterations have been allocated to which threads. The current implementation

is based on a generic linked list library and appears to have overheads that vary with the the number of iterations allocated in a contiguous block to each thread.

While this has little effect on the NAS benchmarks, applications with many small short may be affected by this overhead. It can be significantly reduced via simple compiler and runtime optimizations such as 1. optimizing the iteration accounting and 2. performing compiler transformations to block the parallel for loop so that each iteration performs multiple original iterations and thus performs more useful work for each use of the accounting function.

- **Dynamic:** same as static except that the thread to iteration mapping is done dynamically, based on which threads are currently free. Its performance characteristics are the same as for Static.
- **Guided:** the first half of the iterations are allocated evenly among threads at the start of the loop and subsequent allocations shrink exponentially in size until a minimum iteration chunk size `cksz` is reached. The overheads for *Original* and *C<sup>3</sup>NoPChkpt* are very similar. The overhead for *C<sup>3</sup>PChkpt* is higher but stable for all chunk sizes since the the sizes of the iteration blocks are similar for different chunk sizes.

### 3.9.3.3 Array

The Array benchmarks measure the cost of privatizing otherwise shared variables upon entry into a parallel region (i.e. creating a copy of the variable for each thread). The results of the Array benchmarks for Linux/IA64, Tru64/Alpha and Solaris/Sparc are shown in Table 3.24, Table 3.25 and Table 3.26, respectively.

Table 3.21: Linux/IA64 Scheduling Microbenchmarks (times in  $\mu\text{sec}$ )

<b>Static</b>	<i>No Chunk Size</i>	<i>1</i>	<i>2</i>	<i>4</i>	<i>8</i>	<i>16</i>
<i>Original</i>	0.015	0.019	0.02	0.017	0.017	0.016
<i>C<sup>3</sup> No Checkpoint</i>	0.017	0.02	0.021	0.019	0.018	0.017
<i>C<sup>3</sup> Checkpoint</i>	4.13	7.7	4.8	3.2	2.9	2.8
		<i>32</i>	<i>64</i>	<i>128</i>		
<i>Original</i>		0.016	0.016	0.015		
<i>C<sup>3</sup> No Checkpoint</i>		0.017	0.017	0.017		
<i>C<sup>3</sup> Checkpoint</i>		2.9	3.4	4.2		

<b>Dynamic</b>	<i>1</i>	<i>2</i>	<i>4</i>	<i>8</i>	<i>16</i>	<i>32</i>	<i>64</i>	<i>128</i>
<i>Original</i>	0.29	0.13	0.07	0.048	0.038	0.034	0.032	0.031
<i>C<sup>3</sup> No Checkpoint</i>	0.29	0.13	0.069	0.047	0.037	0.032	0.03	0.029
<i>C<sup>3</sup> Checkpoint</i>	8.6	4.9	3.3	2.9	2.8	2.9	3.4	4.2

<b>Guided</b>	<i>1</i>	<i>2</i>	<i>4</i>	<i>8</i>	<i>16</i>	<i>32</i>
<i>Original</i>	0.29	0.26	0.25	0.2	0.15	0.09
<i>C<sup>3</sup> No Checkpoint</i>	0.29	0.26	0.25	0.2	0.15	0.087
<i>C<sup>3</sup> Checkpoint</i>	2.94	2.9	2.8	2.7	2.7	2.7

Table 3.22: Tru64/Alpha Scheduling Microbenchmarks (times in  $\mu\text{sec}$ ) (90 runs each)

<b>Static</b>	<i>No Chunk Size</i>	<i>1</i>	<i>2</i>	<i>4</i>	<i>8</i>	<i>16</i>
<i>Original</i>	0.013	0.020	0.017	0.021	0.019	0.017
<i>C<sup>3</sup> No Checkpoint</i>	0.018	0.027	0.023	0.024	0.022	0.021
<i>C<sup>3</sup> Checkpoint</i>	1.2	19	11	6.74	4.5	2.9
		<i>32</i>	<i>64</i>	<i>128</i>		
<i>Original</i>		0.017	0.018	0.019		
<i>C<sup>3</sup> No Checkpoint</i>		0.021	0.021	0.021		
<i>C<sup>3</sup> Checkpoint</i>		2.0	1.5	1.3		

<b>Dynamic</b>	<i>1</i>	<i>2</i>	<i>4</i>	<i>8</i>	<i>16</i>	<i>32</i>	<i>64</i>	<i>128</i>
<i>Original</i>	0.72	0.47	0.35	0.27	0.22	0.11	0.038	0.030
<i>C<sup>3</sup> No Checkpoint</i>	0.73	0.54	0.40	0.34	0.24	0.11	0.056	0.051
<i>C<sup>3</sup> Checkpoint</i>	7.1	6.3	5.3	4.2	2.95	1.9	1.4	1.2

<b>Guided</b>	<i>1</i>	<i>2</i>	<i>4</i>	<i>8</i>	<i>16</i>	<i>32</i>
<i>Original</i>	0.070	0.067	0.063	0.061	0.057	0.043
<i>C<sup>3</sup> No Checkpoint</i>	0.086	0.083	0.079	0.078	0.072	0.059
<i>C<sup>3</sup> Checkpoint</i>	1.8	1.7	1.6	1.5	1.4	1.4

Table 3.23: Solaris/Sparc Scheduling Microbenchmarks (times in  $\mu\text{sec}$ )

<b>Static</b>	<i>No Chunk Size</i>	<i>1</i>	<i>2</i>	<i>4</i>	<i>8</i>	<i>16</i>
<i>Original</i>	0.0026	0.14	0.080	0.044	0.026	0.015
<i>C<sup>3</sup> No Checkpoint</i>	0.047	0.19	0.11	0.072	0.053	0.041
<i>C<sup>3</sup> Checkpoint</i>	2.4	19.4	9.3	4.9	3.3	2.7
		<i>32</i>	<i>64</i>	<i>128</i>		
<i>Original</i>		0.010	0.0075	0.0065		
<i>C<sup>3</sup> No Checkpoint</i>		0.036	0.033	0.033		
<i>C<sup>3</sup> Checkpoint</i>		2.4	2.4	2.51		

<b>Dynamic</b>	<i>1</i>	<i>2</i>	<i>4</i>	<i>8</i>	<i>16</i>	<i>32</i>	<i>64</i>	<i>128</i>
<i>Original</i>	5.7	3.0	1.6	0.67	0.34	0.14	0.090	0.078
<i>C<sup>3</sup> No Checkpoint</i>	5.7	3.1	1.6	0.67	0.33	0.13	0.073	0.063
<i>C<sup>3</sup> Checkpoint</i>	18.3	9.7	5.1	3.5	2.8	2.5	2.4	2.4

<b>Guided</b>	<i>1</i>	<i>2</i>	<i>4</i>	<i>8</i>	<i>16</i>	<i>32</i>
<i>Original</i>	0.27	0.25	0.21	0.16	0.14	0.10
<i>C<sup>3</sup> No Checkpoint</i>	0.17	0.15	0.14	0.12	0.10	0.078
<i>C<sup>3</sup> Checkpoint</i>	2.6	2.5	2.5	2.4	2.4	2.3

Table columns correspond to different sizes of the variable being privatized, growing in powers of 3. We report the difference in the cost of a parallel region with privatization and a regular parallel region with no privatization.

- **Private:** measures the cost of using the `private` clause, which privatizes shared local variables. *Original* and *C<sup>3</sup>NoPChkpt* have essentially the same cost, since *C<sup>3</sup>* does not transform parallel regions without checkpoints. *C<sup>3</sup>PChkpt*, shows significantly larger costs because of *C<sup>3</sup>*'s reimplementa-tion of private variables (as discussed in Section 3.8.2.4), which appears to be less efficient than the native implementation. *C<sup>3</sup>* converts the body of a parallel region into a function and passes the privatized variables by value into it, thus creating a private copy of the variable on each thread's stack. This appears to be less efficient than whatever the native system is doing and optimizing this is a useful direction in which to focus our future efforts. On Tru64/Alpha it appears that `private` has a negative cost under *Original* and *C<sup>3</sup>NoPChkpt* but this is very close to 0  $\mu$ s and is almost certainly due to noise.
- **FirstPrivate:** like **Private**, but the `firstprivate` clause initializes each privatized copy of a variable with its pre-parallel region value. Again, *Original* and *C<sup>3</sup>NoPChkpt* have the same cost. *C<sup>3</sup>PChkpt*'s cost is higher but as the same as for the **Private** benchmark since the above implementation ensures `firstprivate` semantics. Note that the cost of `firstprivate` is constant for small variable sizes and increases linearly for large sizes. This is the cost of variable initialization copies.

Table 3.24: Linux/IA64 Array Microbenchmarks (times in  $\mu\text{sec}$ )

<b>Private</b>	<i>1</i>	<i>3</i>	<i>9</i>	<i>27</i>	<i>81</i>	<i>243</i>	<i>729</i>
<i>Original</i>	0.02	0.03	0.01	0.02	0.04	0.03	0.01
<i>C<sup>3</sup> No Checkpoint</i>	0.05	0.06	0.02	0.02	0.04	0.05	0.04
<i>C<sup>3</sup> Checkpoint</i>	9.7	14.2	13.8	12.9	12.6	14.2	15.2
	<i>2187</i>	<i>6561</i>	<i>19683</i>	<i>59049</i>			
<i>Original</i>	0.01	0.03	0.04	-0.001			
<i>C<sup>3</sup> No Checkpoint</i>	0.09	0.07	0.06	0.07			
<i>C<sup>3</sup> Checkpoint</i>	16	16.4	42	43			
<b>Firstprivate</b>	<i>1</i>	<i>3</i>	<i>9</i>	<i>27</i>	<i>81</i>	<i>243</i>	<i>729</i>
<i>Original</i>	0.02	0.07	0.05	0.07	0.1	0.2	0.5
<i>C<sup>3</sup> No Checkpoint</i>	0.08	0.07	0.2	0.2	0.1	0.2	0.6
<i>C<sup>3</sup> Checkpoint</i>	9.5	13.7	13.5	10.6	11.2	13.4	14.6
	<i>2187</i>	<i>6561</i>	<i>19683</i>	<i>59049</i>			
<i>Original</i>	1.8	6	14.9	48			
<i>C<sup>3</sup> No Checkpoint</i>	1.9	6	15	49			
<i>C<sup>3</sup> Checkpoint</i>	15.4	15.9	42	43			
<b>Copyin</b>	<i>1</i>	<i>3</i>	<i>9</i>	<i>27</i>	<i>81</i>	<i>243</i>	<i>729</i>
<i>Original</i>	2.5	2.5	2.5	2.6	2.7	2.6	2.8
<i>C<sup>3</sup> No Checkpoint</i>	2.6	2.5	2.6	2.6	2.7	2.7	3.0
<i>C<sup>3</sup> Checkpoint</i>	11.8	17.3	17.8	15.4	15.7	18.7	22
	<i>2187</i>	<i>6561</i>	<i>19683</i>	<i>59049</i>			
<i>Original</i>	3.8	6.8	17	49			
<i>C<sup>3</sup> No Checkpoint</i>	4.0	7.2	17	49			
<i>C<sup>3</sup> Checkpoint</i>	28	46	148	437			

Table 3.25: Tru64/Alpha Array Microbenchmarks (times in  $\mu\text{sec}$ ) (90 runs each)

<b>Private</b>	<i>1</i>	<i>3</i>	<i>9</i>	<i>27</i>	<i>81</i>	<i>243</i>	<i>729</i>
<i>Original</i>	-0.07	-0.08	-0.07	-0.07	-0.06	-0.07	-0.09
<i>C<sup>3</sup> No Checkpoint</i>	-0.06	-0.04	-0.06	-0.07	-0.07	-0.05	-0.03
<i>C<sup>3</sup> Checkpoint</i>	4.1	4.4	6.7	2.01	0.68	7.9	9.0
	<i>2187</i>	<i>6561</i>	<i>19683</i>	<i>59049</i>			
<i>Original</i>	-0.10	-0.07	-0.007	-0.06			
<i>C<sup>3</sup> No Checkpoint</i>	-0.02	-0.02	-0.02	0.07			
<i>C<sup>3</sup> Checkpoint</i>	12	13	283	286			
<b>Firstprivate</b>	<i>1</i>	<i>3</i>	<i>9</i>	<i>27</i>	<i>81</i>	<i>243</i>	<i>729</i>
<i>Original</i>	-0.07	-0.03	0.04	0.09	0.2	0.4	1.3
<i>C<sup>3</sup> No Checkpoint</i>	-0.04	-0.06	-0.02	0.07	0.2	0.4	1.2
<i>C<sup>3</sup> Checkpoint</i>	2.5	2.1	3.9	-0.04	-0.63	5.7	5.9
	<i>2187</i>	<i>6561</i>	<i>19683</i>	<i>59049</i>			
<i>Original</i>	3.8	40	73	219			
<i>C<sup>3</sup> No Checkpoint</i>	3.5	23	73	222			
<i>C<sup>3</sup> Checkpoint</i>	8.3	8.7	283	287			
<b>Copyin</b>	<i>1</i>	<i>3</i>	<i>9</i>	<i>27</i>	<i>81</i>	<i>243</i>	<i>729</i>
<i>Original</i>	3.0	2.5	2.6	2.5	2.7	2.8	3.6
<i>C<sup>3</sup> No Checkpoint</i>	2.5	2.5	2.6	2.6	2.7	2.9	3.6
<i>C<sup>3</sup> Checkpoint</i>	17	16	13.2	10.1	10.8	21	25
	<i>2187</i>	<i>6561</i>	<i>19683</i>	<i>59049</i>			
<i>Original</i>	6.9	25	74	224			
<i>C<sup>3</sup> No Checkpoint</i>	7.3	25	75	225			
<i>C<sup>3</sup> Checkpoint</i>	44	80	1346	2925			

Table 3.26: Solaris/Sparc Array Microbenchmarks (times in  $\mu\text{sec}$ )

<b>Private</b>	<i>1</i>	<i>3</i>	<i>9</i>	<i>27</i>	<i>81</i>	<i>243</i>	<i>729</i>
<i>Original</i>	8.03	8.0	8.0	8.1	8.1	8.1	8.1
<i>C<sup>3</sup> No Checkpoint</i>	8.06	8.0	8.0	8.0	8.1	8.0	8.0
<i>C<sup>3</sup> Checkpoint</i>	105	108	106	99	97	105	114
	<i>2187</i>	<i>6561</i>	<i>19683</i>	<i>59049</i>			
<i>Original</i>	8.0	8.1	8.1	8.1			
<i>C<sup>3</sup> No Checkpoint</i>	8.1	8.1	8.1	8.1			
<i>C<sup>3</sup> Checkpoint</i>	114	136	508	490			
<b>Firstprivate</b>	<i>1</i>	<i>3</i>	<i>9</i>	<i>27</i>	<i>81</i>	<i>243</i>	<i>729</i>
<i>Original</i>	8.9	9.0	9.3	9.5	12.9	23	53
<i>C<sup>3</sup> No Checkpoint</i>	9.1	9.2	8.9	9.0	12.8	23	53
<i>C<sup>3</sup> Checkpoint</i>	107	110	107	100	97	109	114
	<i>2187</i>	<i>6561</i>	<i>19683</i>	<i>59049</i>			
<i>Original</i>	168	493	1496	4644			
<i>C<sup>3</sup> No Checkpoint</i>	168	525	1588	4659			
<i>C<sup>3</sup> Checkpoint</i>	112	123	556	483			
<b>Copyin</b>	<i>1</i>	<i>3</i>	<i>9</i>	<i>27</i>	<i>81</i>	<i>243</i>	<i>729</i>
<i>Original</i>	11.1	11.3	11.3	11.4	15.3	24.4	53
<i>C<sup>3</sup> No Checkpoint</i>	11.8	11.8	11.9	11.8	15.5	24.7	52
<i>C<sup>3</sup> Checkpoint</i>	112	118	114	107	110	126	157
	<i>2187</i>	<i>6561</i>	<i>19683</i>	<i>59049</i>			
<i>Original</i>	138	396	1195	3449			
<i>C<sup>3</sup> No Checkpoint</i>	136	391	1182	3397			
<i>C<sup>3</sup> Checkpoint</i>	252	573	4436	9736			

- `Copyin`: similar to `firstprivate`, but applies to global variables (marked `threadprivate`; by default OpenMP global variables are shared). *C*<sup>3</sup> Reimplements `threadprivate` variables by `malloc`-ing a copy for each thread and Reimplements `copyin` via `memcpy`. This appears to be slower than using the call stack or whatever the native OpenMP implementations are doing.

### 3.9.4 Discussion

When we began this work, we invested considerable time in refining the generic protocol described in Section 3.4 because we thought that the execution of the protocol would increase the running time of the application significantly. Indeed, much of the literature on fault-tolerance focuses on protocol optimizations such as reducing the number of messages required to implement a given protocol.

Our experiments showed that the overheads are largely due to other factors, summarized below.

- The performance of some codes is sensitive to the memory allocator, with memory locality induced by the allocator playing a dominant role in the application's performance.
- The instrumentation of code to enable state-saving as described in Section 3.8.2.7 prevents certain compiler optimizations, resulting in higher checkpoint-free overheads in codes like `radix` on Tru64/Alpha and MG on Linux/IA64. This is relatively easy to fix by using a `getcontext/setcontext`-based stack saving mechanism (Section 3.2 of [139]).
- For codes that produce large checkpoint files, the time to write out these files dominates the checkpoint time. We are exploring compiler analysis

techniques to reduce the amount of saved state.

- $C^3$ 's compiler transformations can make certain OpenMP constructs more expensive. The effect is insignificant for some constructs and very significant for others (e.g. `parallel` and `private`) and varies significantly depending on whether a given construct contains a potential checkpoint location. While these overheads did not seem to have a large effect on the SPLASH-2 and NAS benchmarks run on larger input sizes (the typical case in scientific computing), it may become important for some codes. Fortunately, many of these overheads can be reduced significantly via small design changes.

### 3.10 Summary

This chapter described a generic protocol for checkpointing shared memory applications running on top of any shared memory API, memory model or implementation. The generality and usefulness of this protocol was experimentally verified by (i) applying this protocol to the OpenMP shared memory API and (ii) implementing this adapted protocol in the  $C^3$  system and experimentally evaluating its performance on three different benchmark suites. Since the adaptation of the protocol to OpenMP was done via application-level checkpointing, additional experiments were performed to compare the sizes of checkpoints generated by  $C^3$  to those generated by SLC solutions.

These evaluations show that the protocol presented in this chapter is in fact generic and can be adapted to real shared memory APIs. This adaptation may require a significant amount of work to checkpoint the state of the application and to adapt the protocol to novel synchronization constructs that may be available

in a given API (e.g. OpenMP's `ordered` regions). However, despite this effort, this protocol presents a high water mark for simplicity in allowing developers to provide arbitrary applications with rollback restart capability without having to develop a new protocol for every new shared memory API, memory model and implementation. Instead, it allows a single protocol to be used in all situations and adapted to the special features provided by the API at hand.

Indeed, the development or and experiments with the  $C^3$  checkpointer for OpenMP application show that the generic protocol can lead to a viable, efficient checkpointing solution for a complex shared memory API. The experimental results in Section 3.9 show that this approach has a significant amount of portability, making it possible to checkpoint applications running on a variety of Operating Systems, Operating Systems, Instruction Sets and OpenMP implementations. In practice, the variations in the dialects of C that need to be parsed we found to be a larger issue deal with than any system-specific problems. If  $C^3$  were to move to an industrial strength compiler such as ROSE[176] even these difficulties would go away.

Additional experiments have validated the efficiency of application-level checkpointing by showing that the checkpoint sizes generated by  $C^3$  were comparable in size to BLCR, a system-level checkpointer, with any checkpoint size reductions by BLCR easily achievable by  $C^3$  via the addition of few system-specific modules.

In addition to presenting concrete techniques for checkpointing OpenMP applications, the chapter presents a general framework for describing and developing Application-level Checkpointing algorithms. This framework for State Recreation consists of the 4 R's: *Restore*, *Replay*, *Reimplement*, and *Restrict*.

When trying to recreate a given piece of state a checkpointer must go through

a number of options. If it has direct access to read and modify this state, then the checkpointer can simply Restore it. If it does not have direct access but the state can be recreated via a sequence of deterministic operations, then we can use the Replay option by issuing those deterministic operations. If the piece of state was created using non-deterministic operations then Replay cannot be used and it is necessary to Reimplement this piece of state and checkpoint this implementation directly. Finally, if Reimplementation proves too difficult or not possible then we can Restrict the set of applications to only those that either do not use the given piece of state or use it in a restricted fashion.

# Chapter 4

## Future Work

### 4.1 Generic Rollback Rollback Restart for MPI

Chapter 2 presented the RROMP system for checkpointing MPI applications. This system succeeds in one major type of portability: it can provide rollback restart for applications that may use any implementation of MPI. However, the current design supports only two protocols (the coordinated checkpointing protocol from Section 2.5.4 and its variant from [189]) and provides no facilities for adding additional protocols without rewriting RROMP's source code. In contrast, Egida [181] is explicitly designed to support a variety of rollback restart protocols that can be implemented via a scripting language.

This provides a number of benefits:

- Because implementing a published protocol can be done without writing the entire checkpointing infrastructure from scratch, it becomes possible for protocol designers to experimentally evaluate their protocols on real systems.
- Since MPI is the dominant message passing API in HPC, an MPI checkpointer is automatically compatible with a large number of real applications. As such, any protocols implemented as part of an MPI checkpointer can be evaluated with real HPC applications.
- Once multiple protocols are implemented as part of a rollback restart system, it is possible to compare the performance of these protocols to each other in order to empirically understand their performance tradeoffs.

However, Egida has only one major drawback: since it is implemented inside of MPICH, it has poor platform portability like all low-level MPI checkpointing solutions. Thus, while it makes it possible to easily implement a variety of real rollback restart protocols and evaluate/compare them on real systems, the fact that Egida is not used in practice on real HPC systems means that the experimental results will not be relevant to the majority of HPC users.

Given RROMP's success in addressing the platform portability problem, the next major direction in RROMP's development is to make it compatible with arbitrary rollback restart protocols. This would create a single system that allows for easy development and evaluation of rollback restart protocols on arbitrary HPC systems, with real-world MPI applications. While the advantages of this are significant, so are the challenges. The protocols in Section 2.5.4 and in [189] were designed to work with an abstract message passing model that includes collective communication and are in fact the first to directly deal with collective communication. This explicit treatment of collectives is required because RROMP has no access to the underlying point-to-point messages that MPI collective routines are likely composed of. Since previously published protocols were developed for a message passing model that includes only point to point communication, this extension of RROMP will need to run these protocols on an abstraction that includes only the data flows of MPI communication routines. RROMP will then need to translate the actions performed by these protocols relative to these data flows to the actual MPI routines that the data flows correspond to.

The result will be a single rollback restart system that is compatible with any rollback restart protocol and any implementation of MPI. It will allow arbitrary protocols to be implemented and experimentally validated on today's popular HPC

systems, with real-world MPI applications. Furthermore, it will allow the first large-scale experimental comparison of rollback restart protocols that is conducted under controlled conditions: a variety of protocols providing rollback restart for the same real-world applications on the same wide range of real HPC systems. The resulting experimental data is expected to be distilled into an analytic performance model that will make it possible to pick the best rollback restart protocol for any application, running on any platform, based on a few empirical tests of the application and platform to determine their relevant properties.

## 4.2 Checkpointing Hybrid MPI/OpenMP Applications

Chapters 2 and 3 present checkpointing solutions for applications parallelized using the MPI and OpenMP APIs, respectively. While these solutions address the needs of a large fraction of real-world applications, the rise of HPC systems built from clusters of shared memory nodes (either symmetric-multiprocessors or chip-multiprocessors) has resulted in a style of parallel programming that uses both message passing and shared memory in a single application, with message passing being used to communicate across nodes and shared memory used for intra-node communication. While such applications may use a variety of APIs to implement this style of communication, MPI/OpenMP hybrid applications [193] are a popular option that is frequently used in practice.

Prior work on checkpointing has focused exclusively on either message passing or shared memory applications, with no attention paid to hybrid applications. Since Chapters 2 and 3 present checkpointing solutions that work with any implementation of MPI and OpenMP, it is possible to combine them to create a checkpointing solution for hybrid MPI/OpenMP applications. Since MPI works

at the level of processes and OpenMP works at the level of threads, it is possible to use the OpenMP checkpointer to save the state of each process at some points in time (including OpenMP state) and use the MPI checkpointer to perform additional coordination and logging to ensure that MPI state is also saved appropriately. The one complication is the fact that the MPI checkpointing protocols presented in this thesis need to record the outcomes of all non-deterministic events performed by the application during a short period of time and repeat these events non-deterministically on restart. The fact that the shared memory checkpointing protocol requires the application to synchronize using only explicit synchronization operations means that this could be done by recording how resources flow from thread to thread and enforcing these decisions on restart.

### **4.3 Hybrid Checkpointing/Message Logging**

While checkpointing is a very popular technique for fault tolerance, it faces significant limits to its scalability as systems grow to hundreds of thousands to millions of processors. This kind of scale has already been achieved with BlueGene/L [27], which currently holds the size record at 128K processors. Furthermore, systems are expected to grow even larger in the future with systems like the BlueGene/Q and the systems under development for the DARPA High Productivity Computing Systems (HPCS) project. The problem is that while small, fairly localized failures are the common failure mode in large systems, the only way for checkpointing to deal with such small scale failures is to roll back all processors to a recent restart line and restart the entire computation. While this is acceptable for most current systems, as the number of processors increases (and with it the system failure rate), the time to record a single checkpoint will grow above the amount of time between

failures. This will render checkpointing completely useless.

This assertion has been validated by a number of studies. [164] has explored the impact of checkpointing on the peta-flop scale machines of the future and found that just considering the likely cost of taking a single checkpoint (estimated at 15 min), a 100hr computation would execute 2.5 times more slowly than in the failure free case.

[76] has performed detailed simulations that considered multiple factors including the cost of checkpointing, the time between checkpoint start and checkpoint commitment and the amount of time to restart a computation. What they found was that hardware failures present a major problem for system scaling. In particular, systems made up of large numbers of nodes fail so frequently that the optimal checkpointing period can grow so short that the application spends almost all of its time checkpointing. Furthermore, they discovered that the cost of checkpointing was just as important as checkpoint commitment time and restart time from the point of view of performance on large numbers of nodes.

[49] and [50] experimentally compares the Chandy Lamport protocol to three message logging protocols in the context of the MPICH-V fault tolerant MPI implementation. In the absence of failures Chandy Lamport features the best bandwidth, latency on 100Mb Ethernet, Myrinet2000 and SCI and application performance on 32 processors connected via 100Mb Ethernet, although causal logging is not very far behind. However, as the probability of failures rises to 1/2 or 2/3 failures per minute, Chandy Lamport spends all of the application's time in taking and recovering from checkpoints, while the message logging protocols continue operating with very little increased overhead as the probability of failures increases.

These results indicate that because checkpointing (i) saves the entire applica-

tion state to disk and (ii) requires all processes to roll back if only one process has failed, it has a natural limitation point after which it becomes useless. At the same time experiments on message logging systems [49] [50] show that they cause impose significant communication costs. One important problem is that they significantly increase the latency of communication, a very vital network property since it tends to be the primary barrier to achieving performance at very large scales. Another problem is that unlike most checkpointing protocols, message logging protocols deal with criss-crossed restart lines, requiring them to log non-deterministic events. While this can be done efficiently in some contexts, this is not true in many other contexts such as shared memory, where every single read is a non-deterministic event.

Given the scalability problems of checkpointing and the poorer fault-free performance of message logging, there is clearly need for a solution that features the best aspects of both. The idea is to divide the application into groups, where each group is checkpointed while inter-groups communication is message-logged. Thus, if one processor fails, all the processors within its group would need to roll back to a prior checkpoint. However, because communication between this group and others was logged, no other group needs to roll back, resulting in a protocol that scales well with rising system sizes and probability of failure. Furthermore, since scalable applications must feature locality of communication (each process mostly communicates within its neighborhood), the amount of inter-group communication in such applications must be a small fraction of overall communication volume. This property will cause the hybrid protocol to feature significantly smaller message logs than those of pure message logging protocols. While this eliminates one major source of overhead in message logging protocols, the hybrid protocol still needs to log the

outcomes of all non-deterministic events, which can have an overhead of its own.

The hybrid rollback restart protocol is in active development and is expected to be incorporated into the RRAMP system. Experiments on large scale systems such as Bluegene/L and ASCI Purple should then provide detailed information about the performance tradeoffs presented by this protocol.

An alternative approach to rollback restart for very scalable systems is presented in [142], which combines coordinated checkpointing of clusters, where whole-cluster checkpoints are initiated using a quasi-synchronous checkpointing protocol. Since this protocol is specifically tuned to applications running on multiple distant clusters that communicate rarely, it is not appropriate for rollback restart for applications running at a single site or on a single scalable cluster.

#### **4.4 Soft Error Vulnerability Compiler Analysis**

While the primary focus on this thesis is on techniques for recovering a computation following a failure, the problem of detecting failures in real systems is equally, if not more, challenging and exciting. The simplest approach is to assume that the system will fail in a fail-stop [188] manner, where it will simply halt and produce no erroneous output. However, this assumption is too strong for real systems, which fail in a variety of complex that may be very difficult to tell apart from correct operation.

One problem that is emerging today is random bit flips due to cosmic radiation. Such radiation hits the Earth's atmosphere, breaking down into a shower of particles, a large fraction of them neutrons. When these neutrons hit the atoms of electronic devices, they can cause current spikes that can flip data bits in memory and change the outcomes of computations. While some bit flips cause obviously

erroneous behavior such as process crashes, many of them cause silent data corruptions, where the program returns an output that looks valid but is actually incorrect. As feature sizes on modern processors become smaller and voltages become lower, the problem of random bit flips due to cosmic radiation is growing more and more severe, threatening the reliability of the results produced by our computing systems.

Because of its importance and difficulty, the problem of detecting and tolerating complex faults (random errors, hacker attacks, etc.) has been studied for five decades, starting with work by von Neumann on fault tolerant circuits [149] in 1956. Much of this work can be divided into two camps: General and Unintelligent vs Specialized and Intelligent.

**General and Unintelligent:** This style of work ignores the details system being analyzed or protected and provides a simple solution for any system. The best example of this camp is modular redundancy [58], where the same computation is performed on multiple (usually two or three) machines and the results of the sub-computations are compared. While much of this work comes from the Systems, Architecture and Electrical Engineering communities, there is interesting work in the Theory community, ranging from work on fault tolerant circuits [165] to the recent breakthrough of the PCP theorem [38]. The advantage of such solutions is their generality. Because they treat the computation being protected as a black box, they apply to any computation. The drawback is the high cost. Depending on the complexity of the fault type under consideration, the overheads range from around 100% to several hundred percent.

**Specialized and Intelligent:** Solutions in this class focus on making specific computations tolerant to faults. For example, it is possible to detect errors in the

output of any sorting algorithm in linear time [213] if the algorithm is augmented to provide a constant amount of authentication information for each number being sorted. Since sorting is an  $O(n \log(n))$ -time problem, the overhead of this solution asymptotically approaches 0%. While much of the work in this class comes from the Theory community, there is also work from Software Engineering, Systems, Architecture and Electrical Engineering. In general, solutions in this class are limited in the kinds of computations they can deal with but provide highly efficient solutions when they do apply.

#### 4.4.1 Moving Forwards

Given the two styles of prior work in this area, it becomes immediately apparent that there is a large gap in the solution space that has not been addressed. What we are missing are solutions that are both General and Intelligent. In other words, solutions that automatically specialize to the computations being protected while being applicable to all or most computations.

From the start this appears to be quite challenging. Consider the most desirable type of solution: for any application it would automatically generate a checker algorithm that verifies that the application's output corresponds to its input and runs in time asymptotically shorter than the original application. Unfortunately, it can be shown via a fairly simple argument that for all problems in  $P$ , if there exists such a checking algorithm, then  $P \neq NP$ . Thus, if we can show that such checkers generally exist then we will have solved one of the holy grails of computer science. Clearly, this is a difficult problem.

While the vast majority of prior work in this subject falls in the above two camps, some work has attempted to stand in between. The best example of this

is the work on compiler transformations that modify the application to enable it to detect or avoid random faults. These transformations are generally simple. Examples include the following:

- Place checks upon entry into and exit from basic blocks to ensure that jump operations are not corrupted [207].
- Transform the application into an alternative version that does exactly the same thing except that all of its numeric values are multiplied by some constant [157]. The idea is to detect permanent faults in a processor by running both versions on the same processor. (running the same version twice would simply result in the same wrong answer twice)

While this work is promising, it still performs very little analysis of the application (and is therefore fairly unintelligent) and features the high overheads common to the General and Unintelligent class of solutions.

I believe that the state of the art in fault detection can be pushed further by adding more intelligence to the compiler in the hopes of producing a more customized fault tolerance solution. The intended result is that the resulting solution will provide the same levels of reliability at a lower cost. The idea is to create a simple fault model (described in Section 4.4.2) and then use a compiler to analyze how the application will be affected by faults described by this model (Section 4.4.3). This information can be used to apply simple fault tolerance techniques like redundancy or placement of checkpoints in extra-reliable locations in the code (Section 4.4.4). Both the analysis and the transformation components of the system would be implemented inside a source-to-source compiler tool.

### 4.4.2 Fault Model

From the application's point of view random faults can manifest themselves as erroneous processor actions and bit flips in memory. However, for the purposes of research we need to create a hierarchy of increasingly accurate and complex fault models. Work can begin at the simpler side of the hierarchy and gradually proceed to address more complex fault models:

1. Faults are limited to memory bits flips that are limited to data memory (instructions are unaffected) and are uniformly distributed.
2. Same as above except that instead of being uniform, the bit flip distribution depends on the structure of the memory. Relevant distribution information would need to be obtained from the manufacturer, via physically sensitive modelling or via live radiation exposure tests.
3. Same as above except that instruction memory may experience bit flips. This extension will require the analyses to be sensitive to the fact that the application's source code may randomly change over time.
4. Same as above except that incorrect processor operation is factored in. This may include incorrect output to arithmetic or logical operations or incorrect jumps. Simple models like the Architectural Vulnerability Factor [144] may be used to describe how the processor responds to faults or model detailed models may be developed for this purpose.

### 4.4.3 Compiler Analysis

Given a fault model it becomes possible to perform a compiler analysis to determine how faults will affect the application. Since this compiler analysis will be based

on a probability distribution of faults, it will produce a probability distribution describing their effects. This section will outline the analysis for the simplest memory model.

The goal of this analysis is to compute the probability that each variable may have an erroneous value at each point in the application. At the start of the application this probability is 0% for all variables. For each operation that is executed the probability rises for all variables by a small  $\epsilon$  proportional to the amount of time it would take to execute the operation. Whenever the value of one variable is used to compute a new value for another variable, the probability of error flows through the computation as follows:

- In any computation operation  $A = B \otimes C$ , if  $B$  or  $C$  have an error then after the assignment,  $A$  will also have an error. Thus, if  $P_{err}[var]$  is the probability of an error in variable  $var$  at a given point in time, after the assignment  $P_{err}[A] = P_{err}[B] + P_{err}[C] - P_{err}[B] * P_{err}[C]$ .
- Whenever the application encounters an if statement, if the variables used in the test condition are erroneous, the application will execute the wrong side of the if. The result is that a number of variables that were not supposed to have been touched are now overwritten with erroneous values and a number of variables that should have been updated with correct values were not. Thus, the probability of error in these variables must increase by the probability that the processor will execute a given if branch erroneously, taking into account the pieces of code that were the correct counterparts to the wrong branch.

The full analysis propagates error probabilities according to the rules above, inter-

procedurally considering correct and incorrect code paths. Iteration and recursion would need to be handled by taking fixed points over the probability formulas.

The above analysis captures a number of key ideas required for accurately estimating the runtime probability of error in every program variable. However, it also features a number of shortcomings that make it inaccurate:

- Variables are treated as either having an error or not with no attention to their individual bits. However, if two variables are used as inputs to a multiplication, errors in their high-order bits will make a much bigger difference in the output than errors in the low-order bits. Thus the analysis would become more accurate if instead of maintaining error probabilities on a per-variable basis, it maintained these probabilities on a per-bit basis.
- The above analysis ignores the values that variables are likely to have. In practice, variables with some values are more vulnerable to faults than variables with other values. For example, if a variable is a boolean and is set to true (=non-zero), the flipping of any bit from 0 to 1 will not change its value. Thus, the accuracy of above analysis would be increased if it were sensitive to the values that different variables are likely to have.
- The analysis description above does not describe how to estimate the amount of time it takes to execute an operation. This is an interesting research subject in its own right and while early versions of the analysis would use simple approximations, advanced versions would need to use accurate estimates based on how real hardware executes instructions.
- The probability that a given variable is corrupted due to the execution of some operation depends on the probability that this operation will ever get

executed. To compute this probability we need to know the probability that the application's if statements will evaluate to true or false. While the simplest estimate would be to treat this probability as 50%/50%, more accurate estimates would use profiling information from the application's previous runs.

- The above analysis description ignores some language features that would make it more complex. Two examples are arrays and pointers.

#### 4.4.4 Guided Fault Tolerance

The fault vulnerability information computed by the above analysis provides a great deal of information about how the application would react to faults. While this information is not sufficient to generate a checker algorithm for the application's results, it is sufficient to intelligently employ some simple fault tolerance techniques, such as the following, thus making it less vulnerable to random faults.

- Let *entropy* be the the aggregate probability of error of all program variables. If the application's entropy increases significantly between the start and end of a code block, this code block can be replicated and executed it twice, comparing the results at the end. Since the probability of both executions being hit by the same fault is miniscule, the replicated code block will only increase the probability of error by a small amount.
- Since memory that is less vulnerable to errors is more expensive than more vulnerable RAM, instead of equipping the computer exclusively with either type of RAM, it would be more cost effective to use a small amount of reliable and a large amount of unreliable RAM, using the above analysis to determine

what variables should be allocated in the reliable RAM for maximal reliability benefit.

- If additional compiler analyses discover invariants that hold between variables (e.g. linear or polynomial relationships), additional reliability can be provided by periodically testing whether these relationships hold at runtime. This technique can also be used to test invariants that hold true of all programs, as is done with control flow checking. The reliability effects of such tests can be incorporated into the above analysis, registering the error probability of the verified variables as dropping immediately after the test.
- Since many of the fault tolerance techniques described here merely detect random errors without tolerating them, checkpointing can be used to recover the application after an error has been detected. However, this would be useless if the fault occurred before the most recent checkpoint was taken. The probability of this even can be reduced by taking checkpoints only at locations in the source code with low entropy.

## BIBLIOGRAPHY

- [1] <http://www.mpi-forum.org/docs/mpi-20.ps>.
- [2] [http://www.llnl.gov/ASC/platforms/lanl\\_q](http://www.llnl.gov/ASC/platforms/lanl_q).
- [3] <http://www.mpi-forum.org/docs/mpi-10.ps>.
- [4] <http://www.mpi-forum.org/docs/mpi-11.ps>.
- [5] <http://www-unix.mcs.anl.gov/mpi/mpich1>.
- [6] <http://www.mcs.anl.gov/mpi/mpich>.
- [7] [http://www.mpi-softtech.com/mpi\\_pro.php](http://www.mpi-softtech.com/mpi_pro.php).
- [8] <http://www.quadrics.com>.
- [9] <http://www.cs.wisc.edu/zandy/ckpt>.
- [10] <http://www.research.rutgers.edu/edpin/epckpt>.
- [11] <http://cryopid.berlios.de>.
- [12] <http://www.llnl.gov/asci/purple/benchmarks/limited/presta>.
- [13] [http://www.llnl.gov/asci/purple/benchmarks/limited/code\\_list.html](http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html).
- [14] <http://www.netlib.org/hpf>.
- [15] <http://www.openmp.org/drupal/mp-documents/spec25.pdf>.
- [16] <http://phase.hpcc.jp/Omni>.
- [17] <http://phase.hpcc.jp/Omni/benchmarks/NPB>.
- [18] <http://www.nas.nasa.gov/Software/NPB>.
- [19] <http://msdn.microsoft.com/vstudio>.
- [20] Posix 1003.1c-1995.
- [21] The sparc architecture manual version 8.
- [22] Rfc 793: Transmission control protocol. <http://www.ietf.org/rfc/rfc0793.txt>, September 1981.
- [23] Man page collection: Shared memory access (shmem). Technical report, Cray Inc, 2003.
- [24] The UPC language specifications v1.2. Technical report, Lawrence Berkeley National Lab, 2005.

- [25] H. Abdel-Shafi, E. Speight, and J. K. Bennet. Efficient user-level thread migration and checkpointing on windows nt clusters. In *USENIX Windows NT Symposium*, July 1999.
- [26] A. Acharya and B.R. Badrinath. Checkpointing distributed applications on mobile computer. In *International Conference on Parallel and Distributed Information Systems*, 1994.
- [27] NR Adiga, G Almasi, GS Almasi, Y Aridor, R Barik, D Beece, R Bellofatto, G Bhanot, R Bickford, M Blumrich, AA Bright, and J. An overview of the bluegene/l supercomputer. In *IEEE/ACM Supercomputing Conference*, 2002.
- [28] Sarita V. Adve and Mark D. Hill. Weak ordering a new definition. In *International Symposium on Computer Architecture*, pages 2–14, June 1990.
- [29] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *International Symposium on High Performance Distributed Computing*, August 1999.
- [30] A. Agbaria and W. H. Sanders. Mobile mpi programs on computational grids. In *IEEE International Conference on Distributed Computing Systems*, 2005.
- [31] R.E. Ahmed, R.C. Frazier, and P.N. Marinos. Cache-aided rollback rror recovery (carer) algorithm for shared-memory multiprocessor systems. In *International Symposium on Fault-Tolerant Computing Systems*, pages 82–88, 1990.
- [32] G. Almasi, C. Archer, J. G. Castanos, M. G. X. Martorell, J. E. Moreira, W. Gropp, S. Rus, and B. Toonen. Mpi on blue gene/l: Designing an efficient general purpose messaging solution for a large cellular system. In *Euro PVM/MPI workshop*, 2003.
- [33] George Almsi, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, Jos E. Moreira, B. Steinmacher-Burow, and Yili Zheng.
- [34] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and orphan-free message logging protocols. In *Symposium on Fault-Tolerant Computing*, pages 145–154, 1993.
- [35] L. Alvisi and K. Marzullo. Trade-offs in implementing optimal message logging protocols. In *Symposium on the Principles of Distributed Computing*, May 1996.
- [36] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

- [37] J. Anfinson and F. T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Transactions on Computers*, 37(12):1599–1604, 1988.
- [38] Sanjeev Arora. Probabilistic checking of proofs and the hardness of approximation problems. Ph.D. thesis, University of California Berkeley, 1994.
- [39] Eduard Ayguade, Marc Gonzalez, Jesus Labarta, Xavier Martorell, Nacho Navarro, and Jose Oliver. Nanoscompiler: A research platform for openmp extensions. In *European Workshop on OpenMP*, October 1999.
- [40] R. Badrinath, R. Gupta, and N. Shrivastava. Ftop: A library for fault tolerance in a cluster. In *International Conference on Parallel and Distributed Computing and Systems*, November 2002.
- [41] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 2nd edition, November 2004.
- [42] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The midway distributed shared memory system. In *COMPCON*, pages 528–537, February 1993.
- [43] B. Bhargava and S. R. Lian. Independent checkpointing and concurrent roll-back for recovery - an optimistic approach. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 3–12, 1988.
- [44] Andrew D. Birrell. An introduction to programming with c# threads. <http://research.microsoft.com/birrell/papers/ThreadsCSharp.pdf> , 2005.
- [45] Joshua Bloch. <http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html>.
- [46] Anita Borg and Sam Glazer Jim Baumbach. A message system supporting fault tolerance. In *Symposium on Operating Systems Principles*, October 1983.
- [47] George Bosilca, Aurlien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fdak, Ccile Germain, Thomas Hrault, Pierre Lemarinier, Oleg Lodygensky, Frdric Magniette, Vincent Nri, and Anton Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *IEEE/ACM Supercomputing Conference*, November 2002.
- [48] Aurelien Bouteiller, Franck Cappello, Thomas Herault, Geraud Krawezik, Pierre Lemarinier, and Frederic Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *IEEE/ACM Supercomputing Conference*, November 2003.

- [49] Aurlien Bouteiller, Thomas Herault, Graud Krawezik, Pierre Lemarinier, and Franck Cappello. Mpich-v: a multiprotocol fault tolerant mpi. *International Journal of High Performance Computing and Applications*, 20:77–90, Spring 2006.
- [50] Aurlien Bouteiller, Pierre Lemarinier, Thomas Hrault, Graud Krawezik, and Franck Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant mpi. In *International Conference on Cluster Computing*, September 2004.
- [51] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the IEEE International Symposium on Reliability, Distributed Software, and Databases*, page 207215, December 1984.
- [52] Greg Bronevetsky and Bronis R. de Supinski. Formal specification of the openmp memory model. In *International Workshop on OpenMP*, June 2006.
- [53] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. In *Symposium on Principles and Practice of Parallel Programming*, 2003.
- [54] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *Principles and Practice of Parallel Programming (PPoPP)*, 2003.
- [55] Greg Bronevetsky, Martin Schulz, Peter Szwed, Daniel Marques, and Keshav Pingali. Application-level checkpointing for shared memory programs. 2004.
- [56] Gilbert Cabillic, Gilles Muller, and Isabelle Puaut. The performance of consistent checkpointing in distributed shared memory systems. In *Symposium on Reliable Distributed Systems*, 1995.
- [57] Brian Carnes. The smg2000 benchmark code. Available at <http://www.llnl.gov/asci/purple/benchmarks/limited/smg/>, September 19 2001.
- [58] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*, February 1999.
- [59] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
- [60] Y. Chen, J. S. Plank, and K. Li. CLIP: A checkpointing tool for message-passing parallel programs. In *SC97: High Performance Networking and Computing*, San Jose, November 1997.

- [61] Z. Chen, G. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault tolerant high performance computing by a coding approach. In *Symposium on Principles and Practice of Parallel Programming*, June 2005.
- [62] P. Emerald Chung, Woei-Jyh Lee, Yennun Huang, Deron Liang, and Chung-Yih Wang. Winckp: A transparent checkpointing and rollback recovery tool for windows nt applications. In *International Symposium on Fault-Tolerant Computing*, page 220, 1999.
- [63] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *ACM/USENIX Symposium on Networked Systems Design and Implementation*, May 2005.
- [64] Christopher Clark. Xen v3.0 user' manual. Technical report, Computer Laboratory, Cambridge University, 2002.
- [65] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label or a simple optimizing compiler for scheme. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 128–139, New York, NY, USA, 1994. ACM Press.
- [66] Aaron Cohen and Mike Woodring. *Win32 Multithreaded Programming*. O'Reilly and Associated Inc., January 1998.
- [67] William W. Collier. *Reasoning About Parallel Architectures*, 1992.
- [68] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauer, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.
- [69] P. Czarnul, A. Urbaniak, M. Fraczak, M. Dyczkowski, and B. Balcerek. Towards easy-to-use checkpointing of mpi applications within clusterix. In *International Conference on Parallel Computing in Electrical Engineering*, pages 390–393.
- [70] Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon, and Alfredo Goldman. Checkpointing bsp parallel applications on the integrate grid middleware. *Concurrency and Computation: Practice and Experience*, 18(6):567–79, May 2006.
- [71] W. Dieter and Jr. J. Lumpp. A user-level checkpointing library for POSIX threads programs. In *Symposium on Fault-Tolerant Computing Systems (FTCS)*, June 1999.
- [72] Chyi-Ren Dow, Jong-Shin Chen, and Min-Chang Hsieh. Checkpointing mpi applications on symmetric multi-processor machines using smpckpt. *The Journal of Systems and Software*, 13:137–150, 2001.

- [73] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, June 1990.
- [74] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *International Symposium on Computer Architecture*, pages 434–442, June 1986.
- [75] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab’s linux checkpoint/restart. Technical report, Lawrence Berkeley National Laboratory, November 2003.
- [76] Elmootazbellah N. Elnozahy and James S. Plank. Checkpointing for petascale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, April-June 2004.
- [77] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.
- [78] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.
- [79] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, October 1996.
- [80] C. Engelmann and G. A. Geist. Super-scalable algorithms for computing on 100,000 processors. In *International Conference on Computational Science*, pages 313–320, May 2005.
- [81] G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. Dongarra. Extending the mpi specification for process fault tolerance on high performance computing systems. In *International Supercomputing Conference*, June 2004.
- [82] Graham Fagg and Jack Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *EuroPVM-MPI*, pages 346–353, 2000.
- [83] Rohit Fernandes, Keshav Pingali, and Paul Stodghill. Mobile mpi programs on computational grids. In *Principles and Practice of Parallel Programming (PPoPP)*, 2006.

- [84] Juan Fernandez, Eitan Frachtenberg, and Fabrizio Petrini. Bcs mpi: A new approach in the system software design for large-scale parallel computers. In *ACM/IEEE SC2003*, Phoenix, Arizona, November 10–16, 2003. Available from [http://hpc.pnl.gov/people/fabrizio/papers/sc03\\_bcs.pdf](http://hpc.pnl.gov/people/fabrizio/papers/sc03_bcs.pdf).
- [85] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.
- [86] George Fishman. *Monte Carlo*. Springer-Verlag, 1996.
- [87] M. Foster and J.N. Wilson. Pursuing the three ap’s to checkpointing with uclik. In *International Linux System Technology Conference*, October 2003.
- [88] Eitan Frachtenberg, Kei Davis, Fabrizio Petrini, Juan Fernandez, and José Carlos Sancho. Designing parallel operating systems via parallel programming. In *Euro-Par 2004*, Pisa, Italy, August 2004. Available from <http://hpc.pnl.gov/people/fabrizio/papers/europar04.pdf>.
- [89] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Pearson Education, June 1999.
- [90] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European PVM/MPI Users’ Group Meeting*, September 2004.
- [91] Guang R. Gao and Vivek Sarkar. Location consistency - a new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 49(8):798–813, 2000.
- [92] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine. A Users’ Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
- [93] David Gelernter, Nicholas Carriero, Sarat Chandran, and Silva Chang. Parallel programming in linda. In *International Conference on Parallel Processing.*, pages 255–263, 1985.
- [94] Kouros Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, 1991.

- [95] Kouros Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *International Symposium on Computer Architecture*, pages 15–26, May 1990.
- [96] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Supercomputing*, November 2005.
- [97] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, March 1989.
- [98] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, September 1996.
- [99] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. Logical time in distributed computing systems. *Parallel Computing*, 22(6):789–828, 1996.
- [100] William D. Gropp. Mpich2: A new start for mpi implementations. In *EuroPVM/MPI Users Group Conference*, October 2002.
- [101] William D. Gropp and Ewing Lusk. Users guide for mpich, a portable implementation of mpi. Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Lab, February 1999.
- [102] Paulo Guedes, Miguel Castro, and Nuno Neves. The disom distributed shared object memory. In *ACM SIGOPS European Workshop: Matching Operating Systems to Application Needs*, 1994.
- [103] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.
- [104] Hiroshi Harada, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Toshiyuki Takahashi, and Yutaka Ishikawa. Scash: Software dsm using high performance network on commodity hardware and software. In *Workshop on Scalable Shared-memory Multiprocessors*, May 1999.
- [105] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, 1993.
- [106] J. M. D. Hill, S. R. Donaldson, and T. Lanfear. Process migration and fault tolerance of bsplib programs running on a network of workstations. In *EuroPar*, 1998.

- [107] J. M. D. Hill, S. R. Donaldson, and T. Lanfear. Process migration and fault tolerance of bsplib programs running on a network of workstations. In *EuroPar*, 1998.
- [108] J. M. Hlary, A. Mostefaoui, R.H. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 10(7):183190, October.
- [109] J. M. Hlary, A. Mostefaoui, and M. Raynal. Virtual precedence in asynchronous systems: concepts and applications. In *11th Workshop on Distributed Algorithms, WDAG*, 1997.
- [110] Chi Ho. A simple message logging protocol. In *Submitted to PODC*, 2006.
- [111] Jay Hoeflinger and Bronis de Supinski. The openmp memory model. In *International Workshop on OpenMP (IWOMP)*, 2005.
- [112] Chao Huang, Orion Lawlor, and Laxmikant V. Kale. Adaptive mpi. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 306–322, October 2003.
- [113] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kale. Performance evaluation of adaptive mpi. In *Symposium on Principles and Practice of Parallel Programming*, March 2006.
- [114] K. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33:518–528, 1984.
- [115] D.B. Hunt and P.N. Marinos. A general purpose cache-aided rollback error recovery (carer) technique. In *International Symposium on Fault-Tolerant Computing Systems*, pages 170–175, 1987.
- [116] Joshua Hursey. Personal Communication, November 2005.
- [117] Liviu Iftode. Home-based shared virtual memory. Ph.D. thesis, Princeton University, Department of Computer Science, 1998.
- [118] William Pugh Jeremy Manson and Sarita V. Adve. The java memory model. In *Symposium on Principles of Programming Languages (POPL 2005)*.
- [119] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Symposium on Fault-Tolerant Computing*, July 1987.
- [120] Greg Johnson Juan Fernandez Eitan Frachtenberg Jos Carlos Sancho, Fabrizio Petrini. On the feasibility of incremental checkpointing for scientific computing. In *International Parallel and Distributed Processing Symposium*, April 2004.

- [121] Laxmikant V. Kale and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on c++. In *Conference on Object Oriented Programming Systems, Languages and Applications*, October 1993.
- [122] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.
- [123] G. Kavanaugh and W. Sanders. Performance analysis of two time-based coordinated checkpointing protocols. In *Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS)*, 1997.
- [124] Darren J. Kerbyson, Adolfo Hoisie, and Harvey J. Wasserman. Modelling the performance of large-scale systems. *IEEE Proceedings on Software*, 150(4):214–222, August 2003.
- [125] Youngjae Kim, Soyeon Park, and Seung Ryoul Maeng. Practical schemes using logs for lightweight recoverable dsm. In *International Conference on Parallel And Distributed Computing and Systems*, November 2003.
- [126] Angkul Kongmunvattan, Santipong Tanchatchawal, and Nian-Feng Tzeng. Coherence-based coordinated checkpointing for software distributed shared memory systems. In *International Conference on Distributed Computer Systems*, 2000.
- [127] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. Seti@home-massively distributed computing for seti. *IEEE MultiMedia*, 3(1):78–83, January 1996.
- [128] C. Kreitz. The nuprl proof development system, version 5: Reference manual and users guide. Technical report, Department of Computer Science, Cornell University, 2002.
- [129] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [130] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- [131] Leslie Lamport. Fairness and hyperfairness. Technical Report 152, Digital Systems Research Center, April 1998.
- [132] Juan Leon, Allan L. Fisher, and Peter Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical report, School of Computer Science, Carnegie Mellon University, February 1993.

- [133] C.-C. J. Li and W. K. Fuchs. Catch - compiler-assisted techniques for checkpointing. In *International Symposium on Fault Tolerant Computing*, pages 74–81, 1990.
- [134] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [135] Deron Liang, P. Emerald Chung, Yennun Huang, Chandra M. R. Kintala, Woei-Jyh Lee, Timothy K. Tsai, and Chung-Yih Wang. Nt-swift: Software implemented fault tolerance on windows nt. *Journal of Systems and Software*, 71(1-2):127–141, 2004.
- [136] G. Liskov, R. Scheifler, E. F. Walker, and W. Weihl. Orphan detection. In *Symposium on Fault-Tolerant Computing*, pages 2–7, July 1987.
- [137] D. Manivannan and M. Singhal. A low-overhead recovery technique using quasi-synchronous checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.
- [138] D. Manivannan and Mukesh Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, July 1999.
- [139] Daniel Marques. Automatic application-level checkpointing for high-performance computing systems. Ph.D. thesis, Cornell University, Department of Computer Science, 2006.
- [140] F. Mattern. Virtual time and global states in distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1989.
- [141] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, 1994.
- [142] S. Monnet, C. Morin, and R. Badrinath. Hybrid checkpointing for parallel applications in cluster federations. In *International Symposium on Cluster Computing and the Grid*, pages 773 – 782, April 2004.
- [143] C. Morin, Lottiaux R., G. Valle, P. Gallard, D. Margery, J. Berthou, and I. Scherson. Kerrighed and data parallelism: Cluster computing on single system image operating systems. In *IEEE Cluster*, September 2004.
- [144] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *International Symposium on Microarchitecture*, 2003.

- [145] Gary L. Mullen-Schultz. *BlueGene/L: Application Development*. IBM Redbooks, 2005.
- [146] R. Nasika and P. Dasgupta. Transparent migration of distributed communicating processes. In *International Conference on Parallel and Distributed Computing Systems*, August 2000.
- [147] George C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages*, 1997.
- [148] R.H.B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.
- [149] J. Von Neumann. *Automata Studies*, chapter Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components, pages 43–98. Princeton: Princeton University Press, 1956.
- [150] N. Neves, M. Castro, and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Symposium on Principles of Distributed Computing Systems (PDCS)*, 1994.
- [151] N. Neves and W. K. Fuchs. Adaptive recovery for mobile environments. In *IEEE High-Assurance Systems Engineering Workshop*, pages 134–141, October 1996.
- [152] N. Neves and W. K. Fuchs. Using time to improve the performance of coordinated checkpointing. In *International Computer Performance and Dependability Symposium*, pages 282–291, September 1996.
- [153] N. Neves and W. K. Fuchs. Using time to improve the performance of coordinated checkpointing. In *International Computer Performance and Dependability Symposium*, pages 282–291, September 1996.
- [154] J. Nieplocha, RJ Harrison, and RJ Littlefield. Global arrays: A portable shared memory model for distributed memory computers. In *Supercomputing*, pages 340–349, 1994.
- [155] R.W. Numrich and J.K. Reid. Co-array fortran for parallel programming. *Fortran Forum*, 17(2), 1998.
- [156] Scott Oaks and Henry Wong. *Java Threads*. O’Reilly and Associated Inc., 2nd edition, January 1999.
- [157] Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. Ed4i: Error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199, February 2002.

- [158] A. J. Oliner, L. Rudolph, and R. K. Sahoo. Cooperative checkpointing: A robust approach to large-scale systems reliability. International Conference on Supercomputing (ICS) 2006.
- [159] Sudakov O.O., Boyko Yu.V., Tretyak O.V., Korotkova T.P., and Meshcheryakov E.S. Process checkpointing and restart system for linux. *Mathematical Machines and Systems*.
- [160] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. In *Symposium on Operating Systems Design and Implementation*, pages 361–376, December 2002.
- [161] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. An evaluation of memory consistency models for shared-memory systems with ilp processors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1996.
- [162] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. Available at <http://www.netlib.org/benchmark/hpl/>.
- [163] Fabrizio Petrini, Kei Davis, and José Carlos Sancho. System-level fault-tolerance in large-scale parallel machines with buffered coscheduling.
- [164] Ian R. Philp.
- [165] Nicholas Pippenger. On networks of noisy gates. In *IEEE Symposium on Foundations of Computer Science*, pages 30–38, 1985.
- [166] J. Planck, J. Xu, and R. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.
- [167] James S. Plank. Efficient checkpointing on MIMD architectures. Ph.D. thesis, Princeton University, June 1993.
- [168] James S. Plank, Micah Beck, and Gerry Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, Winter 1995.
- [169] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. Technical report, University of Tennessee, August 1994.

- [170] James S. Plank and Kai Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel and Distributed Technologies*, 2(2):62–67, Summer 1994.
- [171] James S. Plank and Kai Li. Performance results of ickp—a consistent checkpointer on the ipsc/860. In *Scalable High-Performance Computing Conference*, page 686693, May 1994.
- [172] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.
- [173] M. Poeppe, S. Schuch, and T. Bemmerl. A message passing interface library for inhomogeneous coupled clusters. In *Workshop for Communication Architecture in Clusters*, April 2003.
- [174] J. Postel. Rfc 768: User datagram protocol. <http://www.ietf.org/rfc/rfc0768.txt>, August 1980.
- [175] Milos Prvulovic, Josep Torrellas, and Zheng Zhang. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, May 2002.
- [176] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. In *Conference on Parallel Compilers*, 2000.
- [177] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, August 2002.
- [178] B. Ramkumar and V. Strumpfen. Portable checkpointing for heterogeneous architectures. In *International Fault-Tolerant Computing Symposium*, pages 58–67, June 1997.
- [179] B. Randell. System structure for software fault tolerance. *Journal of Global Optimization*, 1(2):220–232, 1975.
- [180] S. Rao, S. Husain, L. Alvisi, E. Elnozahy, and A. Del Mel. An analysis of communication-induced checkpointing. In *IEEE Fault-Tolerant Computing Symposium*, 1999.
- [181] Sriram S. Rao. Egida: A toolkit for low-overhead fault tolerance. Ph.D. thesis, University of Texas at Austin, Department of Computer Science, 1999.
- [182] R. Recio, P. Culley, D. Garcia, and J. Hilland. An rdma protocol specification (version 1.0). Technical report, RDMA Consortium, October 2002.
- [183] Fiona J. L. Reid and J. Mark Bull. OpenMP microbenchmarks version 2.0. In *European Workshop on OpenMP*, 2004.

- [184] D. Russel. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, 6(2):183–194, 1980.
- [185] Yogish Sabharwal, Rahul Garg, and Vijay K. Garg. Scalable algorithms for global snapshots in distributed systems. In *International Conference on Supercomputing*, June 2006.
- [186] S. Sahni and G. Vairaktarakis. The master-slave paradigm in parallel computer and industrial settings. *Journal of Global Optimization*, 9:357–377, 1996.
- [187] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.
- [188] Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Programming Languages and Systems*, 1(3):222–238, October 1983.
- [189] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *Supercomputing*, 2004.
- [190] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, 7th edition, 2004.
- [191] P.S. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox Palo Alto Research Center, December 1991.
- [192] R. Sites and R. Witek, editors. *Alpha AXP Architecture Reference Manual*. Digital Press, 2nd edition, 1995.
- [193] Lorna Smith and Mark Bull. Development of mixed mode mpi/openmp applications. In *Workshop on OpenMP Applications and Tools*, July 2000.
- [194] D.J. Sorin, M.M.K. Martin, M.D. Hill, and D.A. Wood. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *International Symposium on Computer Architecture*, May 2002.
- [195] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

- [196] Johny Srouji, Paul Schuster, Maury Bach, and Yulik Kuzmin. A transparent checkpoint facility on nt. In *USENIX Windows NT Symposium*, August 1998.
- [197] Nathan Stone, John Kochmar, Raghurama Reddy, J. Ray Scott, Jason Sommerfield, and Chad Vizino. A checkpoint and recovery system for the pittsburgh supercomputing center terascale computing system. Technical report, Pittsburgh Supercomputing Center, 2001.
- [198] Nathan Stone, John Kochmar, Raghurama Reddy, J. Ray Scott, Jason Sommerfield, and Chad Vizino. A checkpoint and recovery system for the Pittsburgh Supercomputing Center Terascale Computing System. In *Supercomputing*, 2001. Available at [http://www.psc.edu/publications/tech\\_reports/chkpt\\_rcvry/checkpoint-recovery-1.0.html](http://www.psc.edu/publications/tech_reports/chkpt_rcvry/checkpoint-recovery-1.0.html).
- [199] Strom, R. E., D. F. Bacon, and S. A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Symposium on Fault-Tolerant Computing*, pages 42–50, 1988.
- [200] F. Sultan, T.D. Nguyen, and L. Iftode. Scalable fault-tolerant distributed shared memory. In *IEEE/ACM Supercomputing Conference*, November 2000.
- [201] Y. Tamir and C.H. Squin. Error recovery in multicomputers using global checkpoints. In *International Conference on Parallel Processing*, page 3241, August 1984.
- [202] R. Thakur and W. Gropp. Improving the performance of collective operations in mpich. In *EuroPVM/MPI Users Group Conference*, 2003.
- [203] S. Vadhiyar and J. Dongarra. Srs - a framework for developing malleable and migratable parallel software. *Parallel Processing Letters*, 13(2):291–312, June 2003.
- [204] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Supercomputing*, 2000.
- [205] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
- [206] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Software - Practice & Experience*, 28(9):963–979, 1998.
- [207] Rajesh Venkatasubramanian, J.P. Hayes, and B.T. Murray. Low-cost on-line fault detection using control flow assertions. In *International On-Line Testing Symposium*, pages 137–143, July 2003.

- [208] K. Venkatesh, T. Radhakrishnan, and H.F. Li. Optimal checkpointing and local encoding for domino-free rollback recovery. *Information Processing Letters*, 25:295–303, July 1987.
- [209] Yi-Min Wang. Space reclamation for uncoordinated checkpointing in message-passing systems. Ph.D. thesis, University of Illinois, Department of Computer Science, 1993.
- [210] Yi-Min Wang. Consistent global checkpoints that contain a set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, April 1997.
- [211] Yi-Min Wang, Pi-Yu Chung, In-Jen Lin, and W.K. Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546–554, 1995.
- [212] M. Wenzel and S. Berghofer. The Isabelle system manual, 2003.
- [213] Dwight S. Wilson, Gregory F. Sullivan, and Gerald M. Masson. Ieee transactions on computers. *Dwight S. Wilson and Gregory F. Sullivan and Gerald M. Masson*, 44(7):833–847, July 1995.
- [214] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. pages 24–36, June 1995.
- [215] Joachim Worringer and Thomas Bemmmerl. Mpich for sci-connected clusters. In *SCI Europe*, pages 3–11, September 1999.
- [216] Kun-Lung Wu and W. Kent Fuchs. Recoverable distributed shared virtual memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.
- [217] Youhui Zhang, Dongsheng Wang, and Weimin Zheng. Transparent checkpointing and rollback recovery mechanism for windows nt applications. In *ACM SIGOPS Operating Systems Review*, page 220, 1999.
- [218] Youhui Zhang, Dongsheng Wong, and Weimin Zheng. User-level checkpoint and recovery for LAM/MPI. *ACM SIGOPS Operating Systems Review*, 39(3), July 2005.
- [219] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kale. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *Conference on Object Oriented Programming Systems, Languages and Applications*, pages 93–103, September 2004.
- [220] Hua Zhong and Jason Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical report, Department of Computer Science, Columbia University, November 2001.