Fault Tolerance using Stable Memory

Edited by

Brian Coghlan
Trinity College Dublin

Germán Fabregat
LISITT

Authors

Michel Banâtre
IRISA/INRIA
Campus universitaire de Beaulieu
F-35042 Rennes cedex
France
michel.banatre@irisa.fr

David Boyce
ex Bull U.K.
Cherry Tree Lane
Hemel Hempstead
U.K.

Brian Coghlan
Department of Computer Science
Trinity College Dublin
Ireland
brian.coghlan@cs.tcd.ie

Mark Cotton
ex Bull U.K.
Cherry Tree Lane
Hemel Hempstead
U.K.
mcotton@sco.com

Germán Fabregat
Departamento de Informática
Universidad Jaume I, Campus del Penyeta Roja sn
12071 Castellón
España
fabregat@vents.uji.es

Alain Gefflaut
IRISA/INRIA
Campus universitaire de Beaulieu
F-35042 Rennes cedex
France
alain.gefflaut@irisa.fr

Jeremy Jones
Department of Computer Science
Trinity College Dublin
Ireland
jeremy.jones@cs.tcd.ie

Philippe Joubert
ex IRISA/INRIA
Campus universitaire de Beaulieu
F-35042 Rennes cedex
France
philippe.joubert@bull.net

Pete Lee
Department of Computing Science
University of Newcastle
Newcastle upon Tyne, NE1 7RU
U.K.
p.a.lee@newcastle.ac.uk

Antonio Marquez
ETRA
Valencia
España
antonio.marques@dg3.cec.be

Rafael Martínez
Instituto de Robótica
Universitat de València
Hugo de Moncado, 4 Entlo., 46010 Valencia
España
rafael.martinez@uv.es

Isi Mitrani
Department of Computing Science
University of Newcastle
Newcastle upon Tyne, NE1 7RU
U.K.
isi.mitrani@newcastle.ac.uk

Christine Morin
IRISA/INRIA
Campus universitaire de Beaulieu
F-35042 Rennes cedex
France
christine.morin@irisa.fr

Antonio Pérez
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid
España
aperez@fi.upm.es

Juan José Serrano
Departamento de Ingeniería de Sistemas
Computación y Automática
Universidad Politécnica de Valencia
España
juanjo@aii.upv.es

Axel Wegner
ARTTIC Hamburg GmbH
Katharinenstr. 11,3. Stock
D-20457 Hamburg
Germany
aw@arttic.com

Contributors

Ekaterina Ametistova
ex Department of Computing Science
University of Newcastle
Newcastle upon Tyne, NE1 7RU
U.K.

Vicente Cerverón
Dpto. de Informática y Electrónica
Universitat de València
Doctor Moliner, 50, 46100 Burjassot Valencia
España
vicente.cerveron@uv.es

Henry Vui Chung
ex Department of Computer Science
Trinity College Dublin
Ireland

Marylène Clatin
IRISA/INRIA
Campus universitaire de Beaulieu
F-35042 Rennes cedex
France

Andrew Cockburn
ex Bull U.K.
Cherry Tree Lane
Hemel Hempstead
U.K.

Cornelius Frankenfeld
ex Stollmann GmbH
Hamburg
Germany

A. García
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid
España

Pedro Gil
Departamento de Ingeniería de Sistemas
Computación y Automática
Universidad Politécnica de Valencia
España

Brian Hennessy
ex Department of Computer Science
Trinity College Dublin
Ireland

Dominic Herity
ex Department of Computer Science
Trinity College Dublin
Ireland

Keith Heron
Department of Computing Science
University of Newcastle
Newcastle upon Tyne, NE1 7RU
U.K.

Madhu Kashup
Bull U.K.
Cherry Tree Lane
Hemel Hempstead
U.K.

Danny Keogan
ex Department of Computer Science
Trinity College Dublin
Ireland

M. A. Liébana
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid
España

Gregorio Martín
Instituto de Robótica
Universitat de València
Hugo de Moncado, 4 Entlo., 46010 Valencia
España
gregorio@glup.irobot.uv.es

Paula McGrath
ex Department of Computer Science
Trinity College Dublin
Ireland

L. M. Muñoz
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid
España

Philip O'Carroll
ex Department of Computer Science
Trinity College Dublin
Ireland

Rafael Ors
Departamento de Ingeniería de Sistemas
Computación y Automática
Universidad Politécnica de Valencia
España

Carlos Pérez
Dpto. de Informática y Electrónica
Universitat de València
Doctor Moliner, 50, 46100 Burjassot Valencia
España
carlos.perez@uv.es

L. Prieto
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid
España

Santiago Rodríguez
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid
España

Vicente Santonja
Departamento de Ingeniería de Sistemas
Computación y Automática
Universidad Politécnica de Valencia
España

Andrew Thomas
ex Department of Computing Science
University of Newcastle
Newcastle upon Tyne, NE1 7RU
U.K.

Preface

Stable 1. Firmly established; not to be easily moved, shaken or overthrown; firmly fixed or settled; as, a *stable* government; a *stable* structure. 2. In *physics*, a term applied to that condition of a body in which, if its equilibrium be disturbed, it is immediately restored, as in the case when the centre of gravity is below the point of support. 3. Steady in purpose; constant; firm in resolution; not easily diverted from a purpose; not fickle or wavering; as, a *stable* man; a *stable* character. 4. Abiding; durable; not subject to be overthrown or changed; as, this life is not *stable*. SYN. Fixed, established, immovable, steady, constant, abiding, strong.

Storage 1. The act of storing; the act of depositing in a store or warehouse for safe-keeping; the safe-keeping of goods in a warehouse. 2. The price charged or paid for keeping goods in a store.

Ogilvie's Imperial Dictionary of the English Language, edited by Charles Annandale, 1895

It is not so surprising that after a hundred years the conjunction of the above two words now has a meaning in the context of computer storage. This book describes work on stable storage technology undertaken within the European Union ESPRIT project P5212 (FASST), as well as in the Basic Research Action QMIPS, and will be of interest to both theoreticians and pragmatists.

The focus of the book is the problem of recovering processor failures in shared memory multiprocessors. We propose an architecture designed for transparently tolerating processor failures. The main component of this architecture is *Stable Memory* (SM), which provides a hardware-supported backward error recovery mechanism. This technique copes with standard caches and cache coherence protocols and avoids rollback propagation.

That the FASST project, which suffered more than the usual quota of difficulties, most notably the bankruptcy of its prime contractor, should have engendered more than the usual quota of high quality work, is a continuing source of interest for those that were involved.

Brian Coghlan
Trinity College Dublin

Germán Fabregat
LISITT

Acknowledgements

The Editors would like to thank the Authors for their contributions, the various host institutions and companies for their permission to publish, and the Commission of the European Communitaires for its financial and administrative support. Finally, for the use of editing facilites, the Editors would like to thank Trinity College Dublin, Universidad Jaume I at Campus del Penyeta Roja sn, and Prof.J.Imberger of the University of Western Australia.

Contents

1 Fault Tolerance **19**
1.1 Introduction 20
1.2 Fault Tolerance Issues 20
1.3 Backward Error Recovery in a Shared Memory Environment 22
1.4 A Basic Recovery Protocol for a Shared Memory Environment 24
1.4.1 Definitions 24
1.4.2 The protocol principles 26
1.4.3 A more rigorous approach to the protocol 28
1.5 Summary 30

2 Modelling **31**
2.1 SMP Cache Coherence Protocols 32
2.1.1 Berkeley Cache Coherency Protocol 32
2.2 Analytical Modelling of SMP Caches 33
2.2.1 Protocol Definitions 34
2.2.2 Cache Line States 36
2.2.3 Bus Queue Metrics 39
2.2.4 Performance Metrics 41
2.2.5 Comparison to Numerical Simulation Results 42
2.3 Queueing Models that include Checkpointing and Recovery 45
2.4 Dependability analysis 47
2.4.1 Multiprocessor system without checkpointing and recovery 48
2.4.2 Non-degradable multiprocessor system with checkpointing and recovery 48
2.4.3 Degradable multiprocessor system with checkpointing and recovery 48
2.5 Summary 48

3 FASST Architecture **65**
3.1 The FASST Architecture 66
3.1.1 Basic Features of the Stable Memory 66
3.1.2 Influence of cache coherence protocols 71
3.2 Performance Evaluation 72
3.2.1 Methodology and workload 72
3.2.2 Experimental results 73
3.2.3 Stable memory implementation 75
3.2.4 Dependency management 75
3.3 Summary 75

4 Stable Memory **83**
4.1 FASST Recovery Protocol 84
4.1.1 Stable Memory 84
4.1.2 Dependency management 85
4.1.3 Synchronization 87
4.1.4 Read and Write Commands 90

4.1.5 Behaviour of the processor initiating a commit 91
4.1.6 Behaviour of other processors 93
4.1.7 SM behaviour during recovery operations 95
4.1.8 Atomic operations 95
4.1.9 Rollback due to a processor failure 95
4.1.10 Various primitives used in the protocol description 98
4.2 Stable Memory Hardware 99
4.2.1 Information flow 99
4.2.2 Fast Serial Link 99
4.2.3 Futurebus+ Interface 99
4.2.4 Other considerations 100
4.2.5 Copy-on-write 100
4.2.6 System Interface 109
4.2.7 Initialization phase 110
4.3 Fault tolerance issues 111
4.4 Expected performance 111

5 Processing Units 115
5.1 Fail-stop Processors 116
5.2 Futurebus+ 116
5.2.1 Data Transfer Bus 116
5.2.2 Arbitration bus and system coordination 117
5.2.3 Cache Coherence 118
5.2.4 Reliability 118
5.3 Dual Processing Units 120
5.3.1 System Memory Map 121
5.3.2 Interrupt scheme 121
5.3.3 Arbitration Messages 122
5.4 The Demonstrator 122
5.5 DPU Prototype 122
5.5.1 Intel i486/DX2 Bus Subsystem 125
5.5.2 Comparators and error control 125
5.5.3 Cache memory and controller 125
5.5.4 *Host Bus* subsystem 126
5.5.5 *FB FPLA* EPLD 126
5.5.6 *CSR LMC FPLA* and *CSR Bus* subsystem 127
5.5.7 Futurebus+ interface 129
5.5.8 Support and Miscellaneous Logic 130
5.5.9 Error detection levels 130
5.6 DPU Demonstrator software 131
5.6.1 Event-driven State Machine 132
5.6.2 Application Executor 133
5.7 Evaluation of the dependability of the DPU Demonstrator 134
5.8 Summary 138

6 Secondary Storage 139
6.1 Secondary Storage 140
6.1.1 Disk Arrays 140
6.2 Redundant Arrays of Independent Disks (RAID) 141
6.2.1 Taxonomy of RAID 141
6.2.2 The First RAID Prototype 145
6.3 Stable Disk 145
6.4 RAID Controller for the Stable Disk 147
6.4.1 Information Flow 149
6.4.2 nMR Operation 150

6.5 Stable Memory for the Stable Disk . . . 151
6.5.1 Intra-bank Checkpointing . . . 153
6.5.2 Inter-bank Checkpointing . . . 155
6.5.3 Log-mode Checkpointing . . . 155
6.5.4 Switch-mode Checkpointing . . . 157
6.5.5 Protection Logic . . . 158
6.6 Integration into the FASST Recovery Protocol . . . 160
6.6.1 Dependency management . . . 161
6.6.2 Synchronisation . . . 162
6.6.3 Read and Write Commands . . . 163
6.6.4 Behaviour of the processor initiating a commit . . . 164
6.6.5 Behaviour of other processors . . . 166
6.6.6 Stable device behaviour during recovery operations . . . 166
6.6.7 Atomic operations . . . 168
6.6.8 Rollback due to a processor failure . . . 168
6.6.9 Various primitives used in the protocol description . . . 169
6.7 Influence on performance . . . 170

7 System Software 175

7.1 System software principles . . . 176
7.1.1 Providing recoverability for the basic model of computation . . . 176
7.1.2 The seed . . . 177
7.1.3 Standard vs non-standard processes . . . 179
7.1.4 Memory management . . . 181
7.2 The Mach Microkernel . . . 181
7.3 OSF1/mk . . . 183
7.3.1 BSD Server . . . 183
7.3.2 Source Tree of OSF1/mk . . . 185
7.3.3 Monoprocessor & Multiprocessor Configurations . . . 186
7.3.4 Compilations: Scripts, Makefiles & Configurations . . . 187
7.3.5 Booting the built OSF1/mk . . . 188
7.3.6 Debugging a Server . . . 188
7.4 System Call Extensions to a Server . . . 190
7.4.1 Adding the `mq open` System Call . . . 191
7.4.2 Adding the `qseek` System Call to Remove the 2GB Filesystem Limit . . . 191
7.5 Driver Extensions to the Microkernel . . . 192
7.5.1 Developing Device Drivers in User-Space . . . 192
7.5.2 The Spy Device `/dev/spy` . . . 192
7.5.3 Dynamically Adding Device Drivers . . . 194
7.5.4 Binding Threads to I/O Capable CPUs . . . 194
7.6 Fault Tolerance Extensions to the Microkernel . . . 195
7.6.1 Fault Tolerant Mechanisms required in the Microkernel . . . 195
7.6.2 Checkpointing . . . 197
7.6.3 Rollback . . . 198
7.6.4 Standard and Non-standard processes . . . 199
7.7 Unresolved Microkernel Issues . . . 202
7.8 Proposal 1 : Boot process handling . . . 204
7.8.1 Mach booting . . . 204
7.8.2 An implementation proposal for boot handling . . . 205
7.9 Proposal 2 : Non-standard I/O process handling . . . 208
7.9.1 I/O handling in Sequoia . . . 208
7.9.2 Reliable I/O Design Principles . . . 209
7.9.3 Device management in the Mach microkernel . . . 213
7.9.4 A first implementation proposal for reliable I/O . . . 218

7.9.5 A second implementation proposal for reliable I/O 226
7.9.6 Comments 229

List of Figures

1.1 A typical shared memory architecture . . . 20
1.2 Communicating processes and recovery . . . 25

2.1 Berkeley state transition diagram . . . 33
2.2 A multiprocessor system . . . 34
2.3 The cache queue . . . 39
2.4 System power versus number of processors, α = 0.85, γ = 0.8 . . . 43
2.5 System power versus *read* probability, K = 25, γ = 0.8 . . . 43
2.6 System power versus *hit* probability, K = 25, α = 0.5 . . . 44
2.7 System power versus *hit* probability, K = 8, α = 0.95 . . . 44
2.8 Two multiprocessing nodes connected to a single shared memory . . . 45
2.9 Processor Model . . . 46
2.10 Simplified Processor Model . . . 46
2.11 FASST queueing model . . . 49
2.12 Influence of cache size on performance . . . 50
2.13 Influence of program size on performance . . . 50
2.14 Influence of read probability on performance . . . 51
2.15 Influence of the checkpointing rate on performance . . . 51
2.16 Reliability for a multiprocessor system without checkpointing and recovery . . . 52
2.17 Availability for a multiprocessor system without checkpointing and recovery . . . 52
2.18 Influence of the processor and memory fault rates on the performability of a multiprocessor system without checkpointing and recovery . . . 53
2.19 Influence of the number of processors on the performability of a multiprocessor system without checkpointing and recovery . . . 53
2.20 Reliability for a non-degradable (CARER) multiprocessor system with checkpointing and recovery 54
2.21 Availability for a non-degradable (CARER) multiprocessor system with checkpointing and recovery 54
2.22 Influence of processor and memory fault rates on the performability of a non-degradable (CARER) multiprocessor system with checkpointing and recovery . . . 55
2.23 Influence of transient fault coverage on the performability of a non-degradable (CARER) multiprocessor system with checkpointing and recovery . . . 55
2.24 Influence of permanent fault rate on the performability of a non-degradable (CARER) multiprocessor system with checkpointing and recovery . . . 56
2.25 Influence of the number of processors on the performability of a non-degradable (CARER) multiprocessor system with checkpointing and recovery . . . 56
2.26 Reliability for a degradable (FASST) multiprocessor system with checkpointing and recovery . . . 57
2.27 Availability for a degradable (FASST) multiprocessor system with checkpointing and recovery . . 57
2.28 Influence of processor and memory fault rates on the performability of a degradable (FASST) multiprocessor system with checkpointing and recovery . . . 58
2.29 Influence of transient fault coverage on the performability of a degradable (FASST) multiprocessor system with checkpointing and recovery . . . 58
2.30 Influence of permanent fault coverage on the performability of a degradable (FASST) multiprocessor system with checkpointing and recovery . . . 59

2.31 Influence of permanent fault rate on the performability of a degradable (FASST) multiprocessor system with checkpointing and recovery . . . 59
2.32 Influence of the number of processors on the performability of a degradable (FASST) multiprocessor system with checkpointing and recovery . . . 60
2.33 Performability comparison with memory fault rate $\lambda_m = 10^{-6}$ . . . 60
2.34 Performability comparison with memory fault rate $\lambda_m = 10^{-5}$ . . . 61
2.35 Processing power for cache coherent dual bus model using the *Write-Once* protocol . . . 61
2.36 Upper bound model for cache coherent dual bus . . . 62
2.37 Lower bound model for cache coherent dual bus . . . 63

3.1 The FASST architecture . . . 66
3.2 Computing a dependency group . . . 69
3.3 Performance . . . 77
3.4 Application behaviour with 8 processors . . . 78
3.5 Application behaviour with 8 processors . . . 79
3.6 Stable memory implementation . . . 80
3.7 efficiency of dependency management . . . 81

4.1 The FASST architecture . . . 84
4.2 Stable memory structure . . . 85
4.3 Read after write dependency: *Commit* . . . 86
4.4 Read after write dependency: *Rollback* . . . 86
4.5 Write after write dependency: *Commit* . . . 86
4.6 Write after write dependency: *Rollback* . . . 87
4.7 Processor automaton . . . 88
4.8 SM states . . . 89
4.9 Timeout protection . . . 90
4.10 Atomic operation . . . 96
4.11 SM board block diagram . . . 101
4.12 Connecting two boards to build an SM module . . . 102
4.13 Update information flow . . . 103
4.14 Commit information flow . . . 104
4.15 Rollback information flow . . . 105
4.16 Autobahn chipset . . . 106
4.17 Autobahn chipset emulation . . . 107
4.18 Frame format and frames used . . . 108
4.19 SM registers . . . 109
4.20 *Vector* information . . . 110
4.21 SM information flow . . . 113

5.1 CSR Space Address Format . . . 117
5.2 Futurebus+ Cache Coherence Protocol . . . 119
5.3 DPU System Memory Map . . . 121
5.4 Two DPUs in the Demonstrator chassis (the DPUs are the modules with cables attached) . . . 123
5.5 Block Diagram of the DPU . . . 124
5.6 Bus 0, Node 2, CSR memory map . . . 127
5.7 Error detection hierarchy . . . 130
5.8 Photograph of the application execution environment . . . 132
5.9 *ESM* module structure . . . 133
5.10 Macroscopic model . . . 135
5.11 The injection environment . . . 136
5.12 DPU Demonstrator coverages . . . 136
5.13 Error detection coverages versus time . . . 137
5.14 Module halt coverages versus time . . . 138
5.15 System reconfiguration coverages versus time . . . 138

6.1 RAID Level 3 143
6.2 RAID Level 5 144
6.3 Multiple Rank RAID with Duplex Controllers 145
6.4 Stable Disk : RAID Controller and Stable Memory 146
6.5 Stable Disk : RAID Chassis and Corollary host 146
6.6 RAID Controller Block Diagram 147
6.7 Layout of modules on the RAID Controller 148
6.8 Interconnection of RAID Controllers and Stable Memories using the VSBus 150
6.9 Bus Interface/Comparator for nMR Operation 150
6.10 Stable Memory Block Diagram 152
6.11 Organization of VRAM Banks in the Stable Memory 152
6.12 Simplified VRAM Block Diagram 153
6.13 Intra-bank Transfer 154
6.14 Inter-bank Transfer 156
6.15 One Possible VRAM Organisation for Log-mode Checkpointing 157
6.16 Switch-mode Checkpointing State Transition Diagram for a Stable Location 158
6.17 Protection Logic Operation 159
6.18 Stable Disk logical structure 160
6.19 Buffered disk requests 170
6.20 A two-node tandem network model 170
6.21 Value of N such that $W \leq 0.05$ sec. $1/\mu = 0.0001$ 173

7.1 Structure of the system software 176
7.2 The short-term scheduler: specifications 178
7.3 The short-term scheduler implementation 178
7.4 Overview of OSF1/mk software architecture 183
7.5 Software architecture of BSD server 184
7.6 Source tree of OSF1/mk Version 4.1 186
7.7 Execution environment of the second server 189
7.8 Partition Table 189
7.9 Virtual machine 195
7.10 Three dependent processes subject to rollback 196
7.11 Dependency relationships between three dependent processes 196
7.12 Execution of a checkpointing algorithm 198
7.13 Checkpointing and rollback 198
7.14 Two dependent processes that checkpoint at context switches 199
7.15 Two dependent processes that checkpoint at any time 199
7.16 Two dependent standard processes 200
7.17 Dependency relationships between the processes in Figure 7.16 200
7.18 Two dependent processes: S3 is a standard process and NS4 is a non-standard process 200
7.19 A standard and a non-standard process, where dependent processes are checkpointed immediately *after* a dependency is created 201
7.20 A standard and a non-standard process, where processes are checkpointed immediately *before* a dependency is created 201
7.21 Flow of control during a Mach boot 205
7.22 Proposed master-slave synchronization 207
7.23 Corollary architecture 209
7.24 Communications in an I/O operation 210
7.25 General I/O handling 211
7.26 Generic layer data structures 214
7.27 SCSI device specific layer data structures 215
7.28 SCSI device disk specific bottom layer data structures 215
7.29 SCSI device disk read path 216
7.30 Console device specific layer data structures 216

7.31 Console device write path 217
7.32 Console device read path 218
7.33 SM and *NRM* data structures 221
7.34 An I/O operation development for the first implementation proposal 225
7.35 Scheduling and forward error recovery actions 225
7.36 A device port structure 227
7.37 An I/O operation development for the second implementation proposal 227

List of Tables

4.1 Assessment of chip count necessary to implement a SM 100

5.1 Futurebus+ global memory map for the Demonstrator 122
5.2 Demonstrator non-DPU components 123
5.3 *ESM* Events 133
5.4 Injected faults 136
5.5 Median of the latency times in nanoseconds 137

6.1 Historical Rates of Improvement in Disk Technology 140
6.2 Switch-mode Checkpointing Truth Table for a Stable Location 157

Chapter 1

Fault Tolerance

1.1 Introduction[1]

Multiprocessor systems based upon standard microprocessors are becoming ever-more commonplace, providing significant computational power at a fraction of the cost traditionally associated with systems of such power. While multiprocessors with distributed memory have gained much attention due to their theoretical peak performance claims, shared-memory multiprocessors continue to be the focus of development of several manufacturers, primarily due to the ease with which such systems can support traditional computing environments and programming paradigms. Nowadays, shared memory systems span the complete range of computing requirements from the personal workstation up to the supercomputer, and with the advent of hardware for distributed shared memory, shared memory has reclaimed a central architectural role.

The dominant organisation of a typical shared-memory multiprocessor is as shown in Figure 1.1, with a single shared bus used to connect processing elements to the shared memory and peripherals. Caches private to the processing elements together with various flavours of snoopy cacheing protocols minimise the bottleneck effect of the single bus. A discussion of the advantages and disadvantages of such architectures is not the concern of this book, and we merely observe that shared bus systems are likely to continue to be constructed. What is of concern to this book is how such systems can be constructed such that hardware faults affecting the processors in the system can be tolerated so that a reliable processing service can be provided in spite of those processor failures.

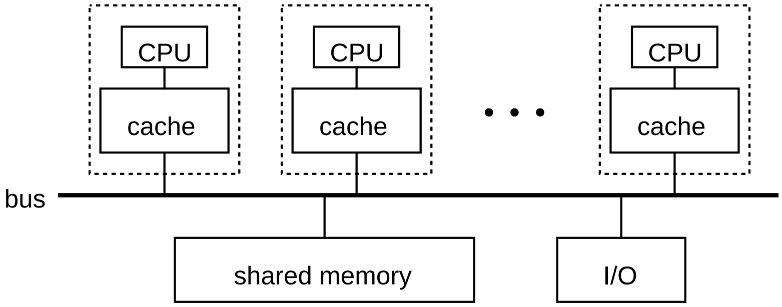


Figure 1.1: A typical shared memory architecture

The need for enhanced reliability is becoming an ever-more critical requirement as computing systems are used for applications where even short breaks in service are unacceptable. Moreover, it is simply infeasible with the complexity and range of present-day software systems to expect that such systems can be enhanced to implement hardware-fault tolerance. What is required is a hardware architecture that can transparently tolerate processor faults, that is, without affecting the executing software and requiring no changes to be made to that software. The presentation of such an architecture for shared memory multiprocessor systems is the primary purpose of this book.

The remainder of this book is organised as follows. First we examine the fundamental problems of providing fault tolerance in a shared-memory multiprocessor, identifying the basic facilities that must be implemented and exemplifying these facilities with examples from some of the fault tolerant multiprocessor systems which are already available commercially, and discussing further the problems of error recovery in multiprocessor systems. Results from simulations of various architectures are also discussed.

We then introduce a fault tolerant architecture which directly supports shared memory semantics. The concept of stable memory, which implements some of the features necessary for transparent fault tolerance, and which is the key novel feature in the architecture, is described in detail, along with associated architectural components.

For simplicity and brevity, the book concentrates on the problems of tolerating processor failures in a shared memory environment and the novel solutions to these problems. Other hardware fault scenarios, such as bus failures, are not covered here, although are clearly important for a complete system.

1.2 Fault Tolerance Issues

The basic principles behind fault tolerance are well understood [Lee et al 90]: a fault in a system will give rise to errors; the starting point for fault tolerance is the detection of an error, and an exception can be raised to signal that

[1] This chapter contributed by Pete Lee, Department of Computing Science, University of Newcastle, Newcastle upon Tyne, NE1 7RU, U.K.

the fault tolerance provisions in the system need to be invoked. These provisions have to:

(a) deal with those errors, in particular to remove errors such that the state is no longer erroneous (error recovery); and

(b) deal with the fault that caused the errors, by identifying its location (fault location), reconfiguring the system to avoid the fault components (fault treatment), and switching the system back to providing its normal operation.

If the above actions are successful, such that the behaviour of the system has not breached the specification of the system, then the system will have successfully tolerated the fault and its effects, and no system failure will be apparent. Preventing system failure is of course the aim of the fault tolerance provisions.

As mentioned previously, this book is concerned with tolerating the faults caused by failures of the microprocessor processors providing the processing power in a multiprocessor with the structure as shown in Figure 1.1, and will therefore concentrate on the application of the above basic principles in this situation. Thus, regarding each processing element as a component of the multiprocessor, we are concerned with providing reliable behaviour in the face of failures affecting these components.

To provide fault tolerance, the first requirement is that the effects of a processor failure are detected. One approach, adopted in the Tandem-16 system [Katzman 78], is to use a single CPU per processing element and to assume that a failure will result in fail-stop behaviour, in that the processing element simply stops if something goes wrong in its logic. The other processors will detect this cessation in service through the absence of the *I'm alive* messages which an active processor regularly broadcasts to all other processors. Note, however, that the Tandem system is not a shared memory multiprocessor, and such single-CPU configurations for a processing element are not the focus of this book.

More active forms of error detection are provided by replication checks where the activity of a CPU is replicated and the outputs from the replicas compared to detect an error. When duplicated CPUs are used, a comparator can detect differences caused if one of the CPUs fails and can raise an exception (or interrupt) to inform the rest of the system of the failure of this processing element such that the fault tolerance actions can be undertaken. This organisation is used in the processing elements of both the Stratus [Wilson 85] and Sequoia [Bernstein 88] fault tolerant systems.

Higher levels of replication, such as using triplicated CPUs in a TMR organisation, can also be used, for instance as in the Tandem S2 system [Jewett 91]. Here a different approach to fault tolerance is being taken as will be seen. In the case of the duplicated CPU discussed previously, a failure of a CPU results in the failure of the processing element of which it is a part: this processing element failure is a fault in the multiprocessor system, and actions elsewhere in the multiprocessor (as will be discussed shortly) have to provide the fault tolerance such that overall system behaviour is not impacted. In contrast, the application of TMR (and higher levels of replication) is simply the application of fault tolerance internal to the processing element such that failures of components within the processing element are never seen by the rest of the system, and for this reason such applications of redundancy are sometimes referred to as masking redundancy. Thus when a CPU fails in a TMR configuration, the divergence of its results from the other two CPUs can be detected by a voter which rejects the *odd man out* (error recovery), ignores the suspect CPU (fault treatment) and passes on the result of the majority to the rest of the system without interruption.

Returning to the situation of the dual-CPU processing element, errors in the system state (i.e. in the global memory) will have to be dealt with if a processing element fails. Errors could have spread in the system by there being a delay between the CPU failure occurring and the processing element actually stopping, during which time erroneous results could have been generated in the shared memory and hence propagated through the system. In the dual-CPU case, there is unlikely to be such a delay and consequent propagation. However, there may still be errors since some of the state of the system will be contained within the failed processing element and this information may be inaccessible. For instance, the contents of the CPU registers and indeed the program counter are all part of the overall system state, and the caches on the processing element may also contain the up-to-date values of some memory locations. Thus, the global memory state may not be consistent with the processing that has been undertaken in the failed processing element, and some form of error recovery will be needed to cope with these errors. Without such error recovery, successful fault tolerance may not be achievable.

Two overall forms of error recovery could be applied: forward error recovery and backward error recovery [Lee et al 90]. Forward error recovery would require the ”patching” up of the system state to fix the problems - for instance, if the failed processing element could be interrogated by another processing element to extract the necessary values, then the system state could be updated appropriately. However, it is unlikely that such an interrogation could take place reliably if a CPU failure has occurred - some of the information may be within the failed chip (e.g. in on-chip caches). Even with duplicated CPUs, it may be difficult to determine which of the pair has failed with the aim of extracting the information from the remaining ”good” CPU. The alternative recovery strategy of backward error recovery requires the state of the whole system to be recovered to a prior known state (simulating the reversal of time, and hence called backward error recovery) such that all of the errors are eradicated. This approach is discussed in detail within this book.

The Stratus system effectively uses a forward error recovery scheme, but avoids the need to interrogate the failed processing element by running a computation simultaneously on two processing elements, each of which contains two CPUs (i.e. on 4 CPUs in total). If one processing element fails, then the other processing element can be used to provide all of the ”internal” values, such that a new processing element can be brought into lock-step and the processing continued (alternatively, the computation can be continued on the single processing element pair with the hope that another failure does not affect that processing element, in which case no error recovery is required). In contrast, the Sequoia system effectively employs backward error recovery, and their scheme is described in the next section.

After error recovery has been carried out, the errors caused by the processing element fault have been dealt with, and so the next stage of fault tolerance is to deal with the fault itself. The location of the fault will be identified by the exception raised in the dual-CPU configuration. If the fault was deemed to be transient (determined, for example, by running diagnostic checks on the faulty processing element), it may be appropriate to permit that processing element to continue to play a part in the system's activity. If, however, the fault is permanent, then that processing element will not be used further, and the computation it was involved in can be restarted on another of the processing elements in the system. If forward error recovery has been used, for instance as in the Stratus system, no processing will have been lost, whereas if backward recovery has been invoked, as in the Sequioa system, some processing will have to be repeated. Note, however, that in a shared memory environment it is a relatively straightforward task to ensure that a computation can be picked up by another processing element - all of the information concerning that computation can be in shared memory and is accessible to all of the processing elements. In a distributed memory situation, as in the Tandem-16 system, this task can be much more complex.

Thus the designer of a fault tolerant multiprocessor is faced with typical engineering trade-offs. Indeed, the different designs taken by Sequoia, Stratus and the Tandem S2 systems suggest that a number of engineering trade-offs are feasible, and that each approach has its place. By adopting triplicated (or higher) levels of redundancy in the processing elements, the need for error recovery can be avoided. However, the cost and difficulties associated with this approach suggest that a design based on duplicated CPUs with provisions for backward error recovery might be more cost-effective. In this book we concentrate on this dual-CPU, backward error recovery approach and on the design of a special form of memory which supports backward error recovery in a shared memory environment. First, though, the basic problems of, and terminology for, backward error recovery in this environment must be discussed so that the facilities that must be provided can be identified.

1.3 Backward Error Recovery in a Shared Memory Environment

The basic functions required for backward error recovery are that a processor can:

(a) establish a recovery point;

(b) recover the state back to that recovery point (roll back); and

(c) commit a recovery point.

The time between the establishment of a recovery point and its eventual commitment is termed a recovery region. A recovery point is thus a point in a computation to which the state can be reset and hence the computation can be restarted from that point. If the establishment of the recovery point preceded the occurrence of a processor failure, then recovery to that recovery point must eradicate all of the potentially erroneous effects of that fault (as discussed in the previous section).

To provide recovery, recovery data must be recorded, for which one of several techniques can be adopted. For example, a checkpoint can be taken when the recovery point is established, that is, a complete copy of the state taken and kept somewhere safe. Since the complete state is likely to be large, and a processor is unlikely to update a significant percentage of its state, more dynamic and optimal facilities can be provided. Shadow paging [Reuter 80] provides a form of incremental checkpointing, by keeping a copy of only those memory pages that have been altered. The recovery cache [Lee et al 80] also provided incremental recording of recovery data.

The Sequoia system makes use of a blocking cache [Bernstein 88] to provide recovery: having established a recovery point, a processor is not permitted to update main memory. Instead all writes are kept local to the processor in a blocking (i.e. non-write-through) cache. If the processor fails, then the state in the main memory represents the state at the recovery point. The commitment of a recovery point by a processor consists of flushing its cache and its internal registers to main memory. Modified data are flushed into two distinct memory modules under processor control in order to handle memory and processor failures.

The CARER architecture [Wu et al 90] makes also use of a blocking cache with the assumption of fault free memory and cache. Assuming that memory is fault free avoids the need for a second memory module for recovery data and hence avoids the loss of time that would be necessary for copying modified data between the two modules. Assuming that caches are fault free limits the work to be done at commit time because blocks residing in cache can be included as recovery data. At commit time, all processor registers are first flushed to the cache and then all modified blocks in the cache are marked *unwritable*. This terminates the commit operation. *Unwritable* blocks belong to the recovery point and have to be written back to memory if they are subsequently modified or replaced (i.e. copy on write).

While the changes a processor makes to memory can be undone by state restoration techniques such as those described above, not all of the manipulatable entities in a system can be recovered. For instance, as discussed in the previous section, the processing element itself may be unrecoverable in that its contents (registers etc.) may not be accessible if that element has failed, so these have to be explicitly recorded when a recovery point is established (e.g. the program counter must be recorded to allow the computation to be restarted from that point). Also, a processor may manipulate other unrecoverable objects, such as peripherals, and the fault tolerant system must cope with the problem of backward recovery in this situation also.

Since recovery data occupies some system resources, it is normal to commit recovery points at some interval, to allow this recovery data to be discarded. For example, in CARER a recovery point is committed each time modified data in the cache has to be replaced, while in the Sequoia system a recovery point has to be committed when the blocking cache of a processor becomes full. For the tolerance of processor faults it is, in general, sufficient to allow a processor to have a single extant recovery point such that the commitment of a recovery point can be synonymous with the establishment of the next recovery point, and thus two distinct operations (establish and commit) are not needed. In a more general situation, for instance if providing software-fault tolerance by recovery blocks [Horning et al 74], nested recovery regions and hence multiple extant recovery points and separate establish and discard operations make sense. As will be seen, even in the simple case it is useful to be able to separate the completion of a recovery region from the commencement of the next.

In a shared memory multiprocessor, there is another complication to recovery that must be dealt with, concerning the parallel processors that will be executing simultaneously and the possible flow of information between these processors via shared memory. Consider the following simple situation: two processors P1 and P2 have separately established recovery points, and P1 has written to a memory location that P2 has subsequently read from and acted upon. Now if P1 fails and backward recovery has to be applied, then it is also necessary to recover P2 to its recovery point which preceded the interaction with P1. Only by recovering P1 and P2 is a consistent system state restored. The recovery points of P1 and P2 which correspond to a consistent state are termed a recovery line [Lee et al 90].

One strategy for identifying a recovery line is to ensure that all processors establish recovery points simultaneously - that is, a system-wide recovery point. If recovery is then required because a processor fails, all processors have to be rolled back. This strategy has the disadvantage of unnecessarily recovering processors when no interactions between a processor and the failing processor have occurred. To avoid this disadvantage effectively requires processors to be recovered independently (rather than globally), and hence requires some other means for solving the problem of interdependencies (i.e. the problem of identifying a recovery line). One method is to avoid the need to identify a recovery line by ensuring that there are no inter-processor dependencies. This can be achieved by not actually providing shared memory (a strategy adopted in the Sequoia architecture) or by committing after each interprocessor interaction in order to remove the dependency (this is essentially what happens in CARER where a processor has to commit its recovery point each time one of its modified blocks in cache is accessed by another processor).

The Sequoia architecture prohibits direct data sharing between processors, leading to significant complications being imposed on the operating system software. While all memory modules can be accessed by all processors, shared data structures must be accessed within explicit critical sections protected by test-and-set locks, and the operating system has to carefully establish and commit recovery points and flush the blocking cache appropriately, to ensure the correct semantics [Bernstein 88].

An alternative solution to identifying recovery lines, which removes the need for the software complexities of the Sequoia approach and the frequent commitments of the CARER approach, is to actually compute a recovery line if recovery is required [Lee et al 90]. In order to do this it is necessary to record inter-processor dependencies which can be used to determine the set of processors which are dependent upon the processor which has failed [Banâtre et al 90a].

It should also be noted that in the Sequoia system it has been necessary to provide custom caches to provide the blocking behaviour. This precludes the use of standard snoopy caches and protocols in such a system. However, the speed of the latest generation of RISC chips is such that their manufacturers provide cache control logic (and chips) as part of their offerings, and it is increasingly difficult (and cost-ineffective) to design custom cacheing systems (and CPUs). Hence, it is desirable that standard processors, caches and cacheing protocols can be used in a shared memory multiprocessor, while still allowing fault tolerance using backward error recovery to be provided.

One may conclude that the concept of *backward error recovery* is now well established as a means of restoring a consistent state to a fault tolerant system should some faults occur [Randell 75]. Several algorithms have been proposed in the literature for providing backward error recovery depending upon the type of faults to be tolerated, the system characteristics, and the fault tolerance strategy.

In a system of communicating processes, should a fault occur, the *recovery control protocol* must determine a set of process states which together constitute a consistent state of the system. Many recovery protocols that assume message passing communication have been proposed in the literature (see for instance [Wood 85, Strom et al 85]). In contrast, the recovery protocol discussed below relies on the fact that communication takes place through shared data, and that the memory itself tracks directly the dependencies between the processors' references to the shared data.

1.4 A Basic Recovery Protocol for a Shared Memory Environment[2]

Before we present a basic recovery protocol for processes communicating through shared data, we must first introduce some definitions and background notions concerning backward error recovery in a system of communicating processes.

1.4.1 Definitions

A *recovery point* is *established* by a process at a point in time at which the state of the process is saved for possible regeneration in the event of recovery action. A process *commits* a recovery point when it no longer requires the capability to initiate recovery action to that point. The period of process activity between the *establishment* of a recovery point and the commitment to it is called the *process transaction* associated with that point (notice that the meaning of the word transaction here should be distinguished from the one which is usually given in transactional systems [Gray 78] where a transaction may refer to a consistency unit preserving some invariant of the system). The most recently established recovery point of a process is said to be *active* or equivalently *current*. A recovery point which cannot possibly be recovered to as a result of a recovery action initiated anywhere in the system is said to be *discardable*. Some of the above definitions are borrowed from [Lee et al 90].

We assume a model of computation of communicating processes where processes implement a succession of *non-nested* transactions, establishing a recovery point immediately on commitment to the preceding one. This is depicted in Figure 1.2 where vertical bars denote the bounds of process transactions. The recovery control management offers the primitive *NewProcessTransaction(p)* for committing the active recovery point and establishing a new recovery point for process p (as a simplification, initialization is not considered). Information flows between processes are assumed to be directed (unidirectional), and are represented by arrows in Figure 1.2 when occurring between distinct processes. It is further assumed that all information sent out by a process is dependent on all information previously received by that process.

[2] This section contributed by Michel Banâtre, Maurice Jégado, Philippe Joubert and Christine Morin, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France

Definition 1 For any two recovery points rp and rp' belonging to processes p and p' respectively, rp is a *direct propagator* to rp' if and only if information flows from p to p' while rp and rp' are the respective active recovery points of the two processes.

As a particular case, a recovery point of a process is a direct propagator to the next recovery point of the same process (for example, in Figure 1.2, recovery point $B.2$ is a direct propagator to $A.2$ and $C.3$ while $A.1$ is a direct propagator to $A.2$). For convenience we will sometimes refer to the propagator relation between process transactions instead of recovery points. A process transaction t_1 is a direct propagator to t_2 if the recovery point established at the beginning of t_1 is a direct propagator to the initial recovery point of t_2.

Definition 2 For any two recovery points rp and rp' belonging to processes p and p' respectively, rp is a *propagator* to rp' if and only if the following holds : Either rp is a direct propagator to rp' or else, recursively, there exists a recovery point rp'' belonging to process p'' such that rp is a direct propagator to rp'' and rp'' is a propagator to rp'

For example, in Figure 1.2, recovery point $B.2$ is a propagator to $A.2$, $C.2$, $C.3$ and $D.2$. As an obvious consequence of the notion of a *propagator*, we will often refer to the recovery *ancestors* and recovery *descendants* of a recovery point rp:

Definition 3 An ancestor recovery point of rp is a propagator to rp. Conversely, if a recovery point rp' is descendant of rp, rp is a propagator to rp'.

For example, in Figure 1.2, recovery points $A.2, B.2, C.2, C.3$ and $D.2$ are ancestors of $D.2$, and in Figure 1.2, recovery points $A.2, C.2, C.3, D.2$ and $B.2$ are descendants of $B.2$. As a particular case, notice that a recovery point is both ancestor and descendant of itself.

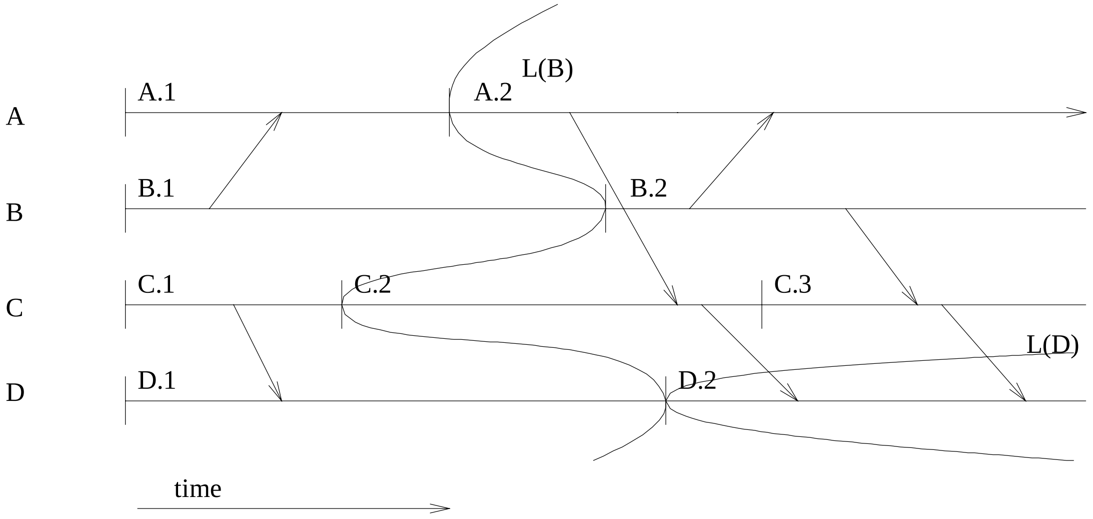


Figure 1.2: Communicating processes and recovery

A recovery protocol must ensure that the system reverts to a consistent state in the event of one (or many) process(es) initiating recovery action. As stated in [Wood 85], *a process initiating recovery must cause recovery of the descendants of its active recovery point* (including the active recovery point itself) in order to reach a consistent state. Another way to say this is that the recovery control protocol must look for a *recovery line* delimiting the boundary of an *atomic activity* [Randell 75].

Definition 4 An atomic action conveys both the meaning of (i) an action which does not *interfere* with its environment, and (ii) a unitary action which has an *all or nothing* effect despite failures, where the first of these properties is referred to as *s_atomicity* (synchronisation) while the second is referred to as *u_atomicity* (unitary). The word atomic alone conveys both meanings.

Recovery lines are depicted as curved lines in Figure 1.2. Not all the processes need be represented on a recovery line, reflecting a situation where one process is not affected by recovery initiated by another. For instance, in Figure 1.2, the recovery line ($L(D)$) associated to process D entails this single process only. Notice also that recovery may in general cause recovery of a process beyond its active recovery point. For instance, in Figure 1.2, recovery initiated by process B will lead to recovery line $L(B)$ thus causing the restoration of process C to recovery point $C.2$ even though $C.3$ is the active recovery point of C.

A process may belong to several recovery lines. For instance, in Figure 1.2, process D belongs to both recovery lines $L(D)$ and $L(B)$. As far as recovery is concerned, we are interested in the recovery line which will lead to the minimal undo of computation ($L(D)$ in this example). This recovery line will be referred to as *the* recovery line of the process.

No information flow crosses from the inside of the recovery line to the outside but the converse is not the case. This requires that the recovery mechanism be capable, in case of recovery, to reproduce the information entering into the domain delimited by the recovery line. For instance, should process D recover in Figure 1.2, the information which has been produced by process C must be available after recovery takes place.

Recovery protocols fall into two broad categories namely *planned* and *unplanned* [Lee et al 90, Randell 75]. Planned (or pessimistic) protocols bound the amount of system activity to be undone in case of recovery at the price of slowing down failure-free computation. In contrast, unplanned (or optimistic) protocols do not slow down failure-free computation but are prone to the so-called *domino effect* (cascading rollbacks) which in the extreme case could invalidate the whole computation in case of recovery.

A recovery protocol must provide the garbage collection of the discardable recovery points that are no longer required to provide backward error recovery capability. While we do not expand on this issue, it should be noted that this might be surprisingly difficult to implement when an unplanned approach to recovery is taken, as illustrated in [Wood 85].

1.4.2 The protocol principles

Assume that processes implement a succession of non-nested transactions. A process may access a local state (the process registers) and a shared state represented in shared memory. Processes communicate through *shared data* of the shared memory. In other words, the significant events produced by a process consist of a trace of accesses to private registers and shared memory.

As a requirement of the protocol, the amount of recovery data must be bounded and limited to a single recovery point per process. Consequently, the protocol must adopt a planned approach to recovery and satisfy the following condition:

$$R : Recovery\ of\ a\ process\ must\ not\ go\ beyond\ its\ active\ recovery\ point$$

From the R condition, we infer:

- **(a)** the domino-effect is prevented, and
- **(b)** it is easy to determine when a recovery point becomes discardable - it is discardable when it is committed.

In order to implement the R condition, the recovery protocol may implicitly establish a recovery point for a process. Notice that this contradicts somewhat the characteristics of the model of computation above since then the decision to establish a new recovery point may be not only explicitly performed by a process but also implicitly by the recovery protocol itself. However, this distinction does not affect the following argument, in which we assume that a process implements a succession of non-nested transactions without further refinement.

In order to give insights into the protocol development, we will use an execution model based on traces and adopt similar notations to those of [Best 82]. An execution U is modelled as a *sequence* $s_0 a_0 s_1 ... s_j a_j s_{j+1} ... a_{u-1} s_u$ where $a_j (0 \leq j < u)$ denotes an action and s_j as well as s_{j+1} denote states. The state space S is defined as the set of mappings from the shared variables space to values. Each action a_j is an *s_atomic* action belonging to a component process $p_i (1 \leq i \leq n)$ denoted $component(a_j)$. An action of p_i is either a write of a value e into a shared variable v denoted $w_i(v, e)$ or a read of a shared variable v denoted $r_i(v)$ or a commitment action denoted $c_i()$. The accesses of a process to its local state are not modelled since these are not relevant as far as communication is concerned. It should be noted that processes proceed asynchronously, and therefore flows of information between them are *non-deterministic* (this model may result from the implementation of an abstract model of computation not detailed here; we may understand non-deterministic flows of information in the model proposed as an implementation property of the abstract model [Gries 81]).

Let s' belonging to S be an arbitrary initial state. The semantics m of the actions performed by a process is a relation $S \times S$ which can be characterized as follows:

- **(a)** $s'\ m(w_i(v,e))\ s$ where $s(v) = e$ and $s(w) = s'(w)$ for all elements w of the state space different to v.
- **(b)** $s'\ m(r_i(v)))\ s$ where $s = s'$ (the read action is supposed to deliver the value $s'(v)$ not modelled here).
- **(c)** $s'\ m(c_i())\ s$ where $s = s'$.

A *projection* of an execution U onto a component process p_i, denoted $projection(p_i, U)$, is obtained by deleting from U all states $s_0 ... s_u$ and all a_j such that $component(a_j) \neq p_i$. We call U a *standard* (or correct) execution if it satisfies the two following properties [Best 82]:

- **($\mathcal{P}$1)** : for all i, $projection(p_i, U)$ is a sequential control sequence of process p_i.
- **($\mathcal{P}$2)** : (s_j, s_{j+1}) belongs to $m(a_j)$ for all j.

A standard execution is said to be *complete* if $projection(p_i, U)$ for all i is a complete control sequence of process p_i. While property $\mathcal{P}1$ captures the control aspect of an execution, property $\mathcal{P}2$ captures the semantic aspects with respect to data (further information can be found in [Best 82]).

Let us turn our attention to recovery now. Should a process p_i roll back when an execution has reached the sequence U, the recovery protocol builds a new sequence W (by undoing the effects of all actions performed within the current transactions of the processes belonging to the recovery line of the process invoking recovery) from which post recovery computation will start. The post recovery computation concatenated with W must be identical to a complete standard execution that would have taken place if recovery did not occur.

To better understand the computation of the recovery line of the process invoking recovery, and more generally into the necessary recovery actions to be taken, let us model the undoing of the effects of a single rolled-back process as the computation of an output *string* U' from an input string U. The string U' is such that $U' = [q_0]b_0[q_1]...[q_j]b_j[q_{j+1}]...[q_{u'-1}]b_{u'-1}[q_{u'}]$ where $[q_j]$ denotes chains of states, and b_j actions. As a particular case, notice that a sequence is a string. The string U' is obtained by:

- **(Step1)** : Erasing from U the actions performed within the current process transaction of the rolled back process (but not the states), and
- **(Step2)** : Appending if necessary a final state to the string denoting the effect of the state restoration applied to the variables of the state space whose values must be recovered by the protocol.

The final output U' of the recovery protocol (after having possibly executed the procedure above for several processes including the process invoking recovery) must be *equivalent* to a standard sequence W satisfying properties $\mathcal{P}1$ and $\mathcal{P}2$. This is defined in the following. Let $W = t_0 b_0 ... t_j b_j t_{j+1} ... t_{u'-1} b_{u'-1} t_{u'}$ such that:

- **(a)** the string of actions b_j of W is identical to the string of actions of U', and
- **(b)** the initial state t_0 of W is equal to the initial state $first(q_0)$ of U'.

The string U' is said to be equivalent to the standard sequence W if:

- **($\mathcal{Q}$1)** : $last([q_{u'}])$ (final state of U') equals $t_{u'}$ (final state of W), and
- **($\mathcal{Q}$2)** : for each read action b_j, $last([q_j])(v)$ equals $t_j(v)$ assuming that v is the variable read by b_j.

If U' is equivalent to the standard sequence W as defined above, it is clear that the output of the recovery protocol is correct; the post recovery computation concatenated with W will be identical to a complete standard execution that would have taken place if recovery did not occur, since the rolled back processes will redo their computation from their current recovery points.

The previous characterization of a correct output of the recovery protocol gives a direct insight into the protocol development. For example, consider the sequence $U = s_0 c_i() s_1 c_j() s_2 w_i(v,e) s_3 r_j(v') s_4 r_i(v') s_5$ and assume that process p_i initiates recovery. Let U' be the string obtained by erasing the actions performed within the current process transaction of p_i and appending a final state s_6 (result of state restoration) such that $s_6(v) = s_0(v)$ and $s_6(w) = s_5(w)$ for all $w \neq v$. More precisely, $U' = [q_0]c_i()[q_1]c_j()[q_2]r_j(v')[q_3]$ where $[q_0] = [s_0], [q_1] = [s_1], [q_2] = [s_2; s_3], [q_3] = [s_4; s_5; s_6]$. The recovery line of p_i entails this single process since U' is obviously equivalent (as defined above) to the standard sequence $W = t_0c_i()t_1c_j()t_2r_j(v')t_3$ where $t_0 = s_0$.

In practice, things might be more difficult than illustrated by the example above, since processes can be dependent. How the protocol deals with this situation is discussed below in a non-formal way by considering in turn the so-called write read and write write dependencies.

1.4.2.1 Write Read dependencies

Let v be a variable written to within the current process transaction of p_i, the first subsequent access to this variable being a read action of p_j within its current transaction. Erasing only the write access from U will not be sufficient to produce a correct string U' since the read action of v by p_j would then not deliver the previous value written to v in U', and hence U' would not be equivalent to the standard sequence W as defined previously (the property $\mathcal{Q}2$ above would not be satisfied). Process p_i is in this case a direct propagator to p_j (or equivalently p_j is *dependent* on p_i), denoted by $p_i \xrightarrow{wr} p_j$, and meaning that rolling back p_i should cause a rollback of p_j. More generally, any process which reads a non-committed value written by p_i is wr dependent on p_i.

Recall that we do not want recovery of a process go beyond its active recovery point (the R condition). In order to ensure this, *the commitment of a process transaction will force the commitment of all its ancestors* (ancestors' recovery points are referred to as potential recovery initiators in [Wood 85]). If this were not the case, an ancestor initiating recovery might require some of its descendants to rollback beyond the current recovery point.

1.4.2.2 Write Write dependencies

Let U' denote the string obtained by erasing from U all actions performed within the process transactions that are descendants of the current recovery point (as explained above) of a process p_i invoking backward error recovery. Let v be a variable which has been written to in U. If the last write to v in U is not erased in U', the value of v in the final state of U' (as given by $Step2$ above) is correct, but elsewhere the value of v is not correct, since U' would not be equivalent to the standard sequence W as defined previously (the property $\mathcal{Q}1$ would not be satisfied). Let $U' = [q_0]...w_i(v, e)...[q_{u'}]$ where $w_i(v, e)$ denotes the last write to v in U' and assume that the last write to v in U has been erased in U'. A correct string could be obtained by appending to U' the result of a state restoration re-establishing the value e of v and repeating this for all variables whose last write to in U has been erased in U'. The difficulty here resides in finding out the value e within the whole history of the values taken by the variable v. How this is achieved is discussed in the following.

Definition 5 A process p is said to be the *active writer* of a variable v if p has been writing to v within its current (active) transaction and v has not been subsequently written to by other processes.

The protocol does not maintain the whole history of a variable but only a *current* value and a *recovery* value. At commitment of a process, the recovery value of a variable is replaced by the current value if the process is the active writer of the variable. Symmetrically, at rollback of a process, the current value of a variable is replaced by the recovery value if the process is the active writer of the variable. *In order to re-establish a valid final state of the string U' above, the protocol ensures that the last write action to v, $w_i(v, e)$, of the string U' is committed and will thus restore the recovery value e of v.* This implies that a process committing its current transaction must force commitment of the active writers of the variables written within the committing transaction, while rollback of a process transaction must cause the rollback of all transactions which have been writing to a variable whose active writer is the rolling back transaction. A simple way to achieve this goal is to record a dependency $p_i \xleftarrow{ww} p_j$ (i.e. $p_j \xrightarrow{ww} p_i$) when a variable is successively written by two processes p_i and p_j within their current process transaction; this dependency will have the same effect as the $\xrightarrow{wr}$ dependency as far as commitment and rollback are concerned. Notice that the $\xrightarrow{ww}$ dependency is a predecessor relation, as opposed to the $\xrightarrow{wr}$ dependency, which captures a successor relation.

1.4.3 A more rigorous approach to the protocol

In this section, we attempt to give some proof arguments of the protocol whose operational principles have been described above. The properties of a standard execution have already been given. We require a weaker property for exceptional sequences than for standard sequences, namely *validity*. An execution U is said to be *valid* if it satisfies the following properties:

- **($\mathcal{P}$'1)** : The same as property $\mathcal{P}1$ above.
- **($\mathcal{P}$'2)** : For all read actions a_j of U, $s_{j+1}(v) = s_j(v)$ if v is the variable read by a_j.
- **($\mathcal{P}$'3)** : Let $U = s_0a_0...a_{u-1}s_u$ followed by an optional state restoration action on behalf of the recovery process (s'_u denoting the final state of U). There exists a standard execution $W = t_0a_0...a_{u-1}t_u$ (satisfying $\mathcal{P}1$ and $\mathcal{P}2$) where t denotes a state such that $t_0 = s_0$ and t_u (final state of W) is identical to s'_u (final state of U).

While $\mathcal{P}2$ captures the exact intermediate states of a standard execution, properties $\mathcal{P}$'2 and $\mathcal{P}$'3 together say that we only need to have a partial knowledge of the intermediate states for checking the validity of an execution. The intuitive rationale behind this is that we wish to be allowed to erase a write action from an execution without being obliged to systematically consider invalid the following sub-sequence, as would be so when assuming property $\mathcal{P}2$. It should be noted that formally, a standard execution is valid but the contrary might not be the case; we will, however, construct exceptional valid sequences in such a way that an exceptional execution will correspond to an equivalent standard execution (in fact the standard execution W described in $\mathcal{P}$'3).

Let U be a valid execution (equivalent to a standard execution) when p_i rolls back. Let $remove(U, p_i)$ be the sequence U from which are deleted all a_js_{j+1} such that $component(a_j) = p_i$, where a_j has been performed within the current process transaction of p_i. Let us attempt to prove that the sequence U' obtained by applying successively $remove$ for all processes that are descendants of the current transaction of p_i (according to both relations $\xrightarrow{wr}$ and $\xrightarrow{ww}$ defined above) is valid, i.e. satisfies properties $\mathcal{P}$'1, $\mathcal{P}$'2 and $\mathcal{P}$'3.

Theorem 1 *The sequence U' satisfies $\mathcal{P}$'1.*

Proof : Since U is valid (satisfying $\mathcal{P}$'1), removing the accesses of the rolled back processes within their current process transactions will lead to $projection(U', p_i)$ being a prefix of $projection(U, p_i)$ for all i, so that $\mathcal{P}$'1 is maintained.
□

Theorem 2 *The sequence U' satisfies $\mathcal{P}$'2.*

Proof : Assume that U' does not satisfy $\mathcal{P}$'2: there exists a substring $s_{k'}r_i(v)s_{k'+1}$ of U' such that $s_{k'+1}(v) \neq s_{k'}(v)$. By construction, $s_{k'+1}(v)$ is the result of the last write to v in U, say by p_j, before the read action r_i by p_i took place. This write has been erased in U' and thus has been performed within the current process transaction of p_j. Two cases arise. First, the read action might have been performed within the current process transaction of p_i, but in that case a dependency $p_j \xrightarrow{wr} p_i$ would have been recorded, and the rollback of p_j would have forced the rollback of p_i and therefore the case cannot arise. Second, the read action might have been performed within a process transaction of p_i which is not the current transaction, but then the write cannot have been erased since the commitment of this process transaction of p_i would have forced the commitment of the process transaction of p_j in which the write would have been performed.
□

Theorem 3 *The sequence U' satisfies $\mathcal{P}$'3.*

Proof : Let W' be the standard sequence built from U' by $\mathcal{P}$'3. Assume that $t_{w'}(v) \neq s'_{u'}(v)$, thereby invalidating $\mathcal{P}$'3. Two cases arise. First the last write to v in U might not have been erased in U', but in that case the state restoration action would not modify v, since the active writer to v would not have been rolled back, and therefore $\mathcal{P}$'3 would be valid, with $s_{w'}(v) = s_{u'}(v) = s'_{u'}(v)$. Second, the last write to v in U might have been erased in U', which would force the removal of all the write operations to v since the last committed write to v due to the relation $\xrightarrow{ww}$. But it is this value of the last committed write which must be the recovery value of v re-established by the state restoration action, since commitment induces the commitment of all ancestors, including the active writer to v, thereby updating the correct recovery value of v.
□

Thus a $p_i \xrightarrow{wr} p_j$ dependency is recorded when p_j reads a variable whose active writer is p_i, while a $p_i \xleftarrow{ww} p_j$ dependency is recorded when p_j writes to a variable whose active writer is p_i (p_j becoming then the new active writer). Rollback of a process p_i will induce a rollback of its descendants according to both relations $\xrightarrow{wr}$ and $\xrightarrow{ww}$; the recovery values will be re-established for those variables whose active writer is a process member of this group. Commitment of a process will induce the commitment of all its ancestors according to both relations $\xrightarrow{wr}$ and $\xrightarrow{ww}$; the recovery values will be logically replaced by their current values for those variables whose active writer is a process member of this group.

1.5 Summary

We now have defined the essence of the FASST Recovery Protocol : a commit *checkpoints* processes as outlined above, and a rollback *restores* checkpointed process state, again as above. All of the above is process oriented, and says nothing about where the processes are executing, as long as each process has a consistent view of the shared memory.

For the simplistic case where a separate processor is dedicated to each process, the above principles may be applied simply by mapping a *process transaction* to a *processor transaction*. We will examine this, and the more realistic case where a multiprocessor may support the execution of an arbitrary number of processes competing for a limited number of available processors, in Chapter 7. For the moment, let us assume that all of the above remains valid if we replace process p_i with processor P_i, where a commit checkpoints *processors* as outlined above, and a rollback restores checkpointed *processor* state, again as above. This is a basic, but very important, assumption of the FASST architecture.

In a symmetric multiprocessor (SMP) all of this is made more interesting by the behaviour of the coherent caches. Before introducing the FASST architecture, we will examine this behaviour, and try to quantify what happens when it is coupled with a recovery protocol.

Chapter 2

Modelling

2.1 SMP Cache Coherence Protocols[1]

A cache system is said to be *coherent* if every read of a memory location returns the value most recently written to that location [Censier et al 78]. In a shared memory multiprocessor where processors access shared memory through private caches, there can be potentially as many copies of the same memory location as there are processors in the architecture. Inconsistencies may occur when several processors access writable shared data. When data is modified, the modification has to be reflected into all the other caches which hold a copy of the data. The unit of information managed by the caches is referred to as a $line$, while a processor accesses a $cell$ (e.g. 4 bytes). Typically a cache line size ranges from 4 to 32 cells.

The protocols for avoiding cache inconsistencies are often referred to as *cache coherence protocols* (the term *cache consistency protocols* can also be found in the literature). These protocols divide into two main classes, *directory-based* and *snoopy*. The less common of the two classes, *directory-based* protocols, rely on a directory structure to identify the location of any cache line and its state. *Snoopy* cache coherence protocols are by far the more common, and rely on the fact that broadcasted bus traffic can be monitored (*snooped on*) by all the caches. Snoopy caches maintain a *tag field* stored along with each loaded line to indicate the line state in each cache. The tag field generally encodes whether the line is modified with respect to shared memory and whether the line is loaded into another cache. Two main classes of snooping cache coherence protocols can be distinguished, depending upon the actions performed by caches when a shared line is modified:

Write Invalidate protocols cause an invalidation message to be broadcast on the bus whenever data potentially present in other caches is updated. All caches snoop these invalidation messages and invalidate their corresponding entry. A further read miss will cause the up-to-date data to be loaded into the cache.

Write Update protocols broadcast the new value whenever data potentially present in other caches is updated. All caches snoop the write and update their copy of the data accordingly.

These protocols mainly differ by their relative hardware cost and their performance in terms of bus traffic generated to maintain coherence (see [Archibald 86] for a survey and performance evaluation of those protocols).

2.1.1 Berkeley Cache Coherency Protocol

The Berkeley coherence protocol [Katz et al 85] was originally designed for the SPUR workstation at the University of California at Berkeley; it is a write invalidate protocol. This protocol introduces the notion of ownership of a line, the owner being responsible for writing the line back to main memory as well as for supplying the line directly to any other cache requesting it. In this protocol, the tag field of a memory line of a given cache can be in one of the following four states (line states are described according to the terminology found in [Sweazey et al 86]) :

Invalid (I) : The cache copy is not up-to-date.

Non-modified Shared (S) : The line has not been modified since it was loaded into this cache. Other caches may also have a copy; one of these copies might be in state O while others must be in state S.

Modified Exclusive (M) : The line is modified with respect to shared memory. No other copy exists. This cache is the owner of the line.

Modified Shared (O) : The line is modified with respect to shared memory. Other caches may have a copy (in state S). This cache is the owner of the line (hence the abbreviation O).

Figure 2.1 depicts the state transition diagram for the Berkeley protocol.

Processor P_i performs a Read Miss If there exists a cache with a copy of the line in state M or O, this cache must supply a copy of the line to the requesting cache and set its state to O. Otherwise the line comes from shared memory. In both cases, the line is loaded in state S in the requesting cache.

[1] This section contributed by Michel Banâtre, Maurice Jégado, Philippe Joubert and Christine Morin, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France

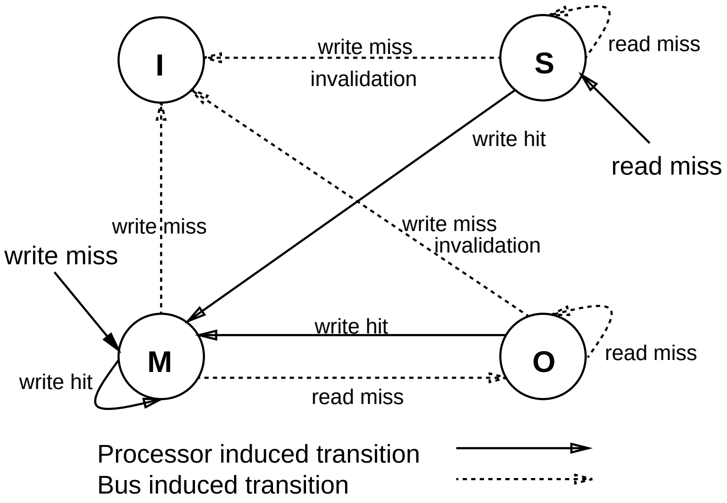


Figure 2.1: Berkeley state transition diagram

Processor P_i performs a Write Hit If the line is already in state M, the write proceeds without delay. Otherwise, (in state S or O) an invalidation signal must be sent on the bus (see figure 2.1). All other caches invalidate their copy if they have one that matches the line address. The line state is changed to M in the originating cache.

Processor P_i performs a Write Miss Like a read miss, the line comes from its owner or from shared memory. All other caches invalidate their copy if any. The line is loaded in state M.

Similar principles to those discussed above in the framework of Write Invalidate protocols also apply to Write Update protocols, such as the Firefly protocol [Thacker et al 88]. In fact, quite a number of both types of snoopy coherence protocols have been proposed and implemented (e.g., [Goodman 84, Fielland et al 84, Rudolph et al 84, Katz et al 85, Archibald 86, Archibald et al 86, Goodman 87]). These range from the simple *Write-Through* protocol [Fielland et al 84, Goodman 87], whereby every write access to a cache line is accompanied by an update of main memory, to the rather complex, such as the *Dragon* and *R4000* protocols [Archibald et al 86, Mirapuri et al 92] (and the *Berkeley* example above), which delegate most of the book-keeping operations to the caches and attempt to minimize memory accesses. The idea behind the latter designs is to exploit the fact that a cache-to-cache transfer is generally faster than a cache-to-memory one.

2.2 Analytical Modelling of SMP Caches[2]

By way of example, let us illustrate the effects of cache behaviour by modelling and evaluating the performance of cache coherence protocols. Two distinct versions will be considered, both resembling the Berkeley protocol but also differing from it in some important respects. These will be referred to as the *Invalidate* and the *Update* protocols. The difference between them lies in the way they handle a *write hit* access to a shared cache line: the former broadcasts a signal which causes all other caches to invalidate their copies, whereas the latter broadcasts the new content of the line and all other caches update their copies. There are non-trivial trade-offs involved in choosing one or the other of the two variants, and the model can be used to assess the effect of various system parameters.

A two-stage approximate analysis, similar to the one introduced in [Greenberg et al 88] in the context of *Write-Through*, *Write-Back* and *Dragon*, is applied to the evaluation of the *Invalidate* and *Update* protocols. First, the behaviour of an individual cache line is modelled by a finite-state Markov process. Its parameters are the number of processors, the ratio of *read* to *write* accesses and the ratio of *hit* to *miss* accesses. The protocols exhibit certain

[2] This section contributed by Ekaterina Ametistova and Isi Mitrani, Department of Computing Science, University of Newcastle, Newcastle upon Tyne NE1 7RU, UK

complications which were not present in the cases examined in [Greenberg et al 88]. Here one has to resort to a fixed-point approximation, since in order to reflect properly the interactions between processors, the generator matrix of the Markov chain needs to depend on its own stationary distribution.

The solution of the cache line model, together with the various access time characteristics of the hardware, provides the relevant bus traffic parameters: rate of bus requests per processor and average service time per request. In the second stage of the analysis, the bus is modelled as a single-server, finite-source queue, in order to determine the system power and other performance measures.

The two protocols are described in section 2.2.1. The approximate analyses of the cache line states and of the bus are presented in sections 2.2.2 and 2.2.3. Section 2.2.5 contains numerical and simulation results, while some generalizations and extensions of the approach are outlined after that.

2.2.1 Protocol Definitions

The system contains K identical processors and their caches (by 'cache' we mean the medium-sized, secondary memory unit associated with a processor, rather than the small, primary one which is really part of the processor itself). These are connected to each other and to the large main memory by means of a bus (Figure 2.2). Exchange of information between processors, or between a processor and main memory, takes place through the caches.

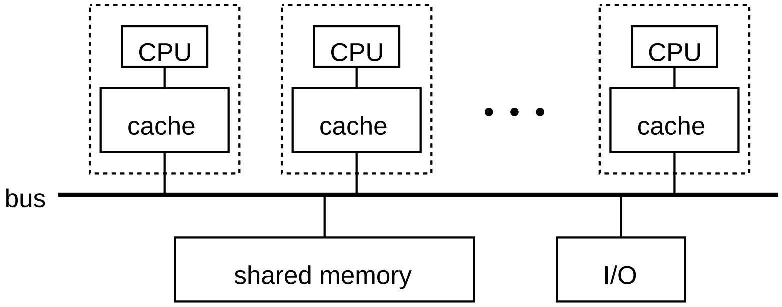


Figure 2.2: A multiprocessor system

Each cache consists of N 'lines'. The size of main memory is M lines, which we shall assume, for simplicity, to be a multiple of the cache size: $M = mN$ for some $m > 0$. There are thus m main memory line addresses which are mapped onto a given cache line address. At any time, one of those m main memory addresses is associated with the cache line. If, when referring to a cache line, a processor accesses the main memory address currently associated with it, then the access is said to be a *hit*; otherwise it is a *miss*. In addition, the access may be a *read* or a *write* (a *load* or a *store*).

When the same main memory address is associated with cache lines in several caches, the latter are said to be *sharing* a line. The content of such a line must be the same in all sharing caches, although it need not necessarily be the same as the corresponding main memory line. So, the accesses by one processor to its cache may have implications for other caches and/or main memory. The precise consequences of each access depend on its nature, on the current state of the line, and on the cache coherence protocol that is employed.

2.2.1.1 The Invalidate Protocol

A cache line may be in one of the following states:

State I : The line is invalid (it does not contain useful information).

State $D(n)$: The line is dirty (main memory does not have an up-to-date copy); n other caches have a copy of that line ($n = 0, 1, \ldots, K-1$), but this cache is its 'owner' and is responsible for saving it should the processor make a *miss* access to it. Also, this cache will supply the line if a processor outside the sharing group accesses it.

State $S(n)$: The line is dirty; n other caches have a copy of it ($n = 1, 2, \ldots, K-1$); this cache is not the owner (in one of the other n caches, the line is in state $D(n)$).

State $C(n)$: The line is clean (main memory has an up-to-date copy); n other caches also hold this line ($n = 0, 1, \ldots, K-1$), but there is no owner. Main memory supplies the line if a processor outside the sharing group accesses it.

It should be noted that the hardware implementation of the protocol does not involve the integer n. A cache line is tagged as being either 'invalid', 'clean exclusive', 'dirty exclusive', 'dirty shared' or 'shared'. The first four of those tags correspond to our states I, $C(0)$, $D(0)$ and $D(n)$ ($n > 0$) respectively; the tag 'shared' may correspond to states $C(n)$ or $S(n)$, for $n > 0$. We need the more detailed state description given above in order to model the evolution of line states, and also to keep track of the different types of bus operations. For example, if a processor makes a *read miss* access to a line and joins a sharing group where all members are in state $C(n)$, then main memory supplies the information and a bus operation of type "main memory to cache" is performed. On the other hand, if the processor joins a sharing group where one member is in state $D(n)$ (the others being in $S(n)$), then a cache supplies the information and a bus operation of type "cache to cache" is performed. Those operations take different times.

It should also be pointed out that, while bus operations involve whole lines (e.g. 16 bytes), a processor can load or store information into part of a line (e.g. 4 bytes). This discrepancy implies that any *miss* access, even a *write* one, requires a new line to be brought into the cache.

An access by a processor to a line in its cache is said to be *local* if it does not involve the bus. Otherwise it is *remote*. All *read hit* accesses to lines in state $C(n)$, $D(n)$ or $S(n)$ are local and do not change the state. *Write hit* accesses to lines in state $C(0)$ or $D(0)$ are also local and the resulting state is $D(0)$. All other accesses are remote and have the following effects:

(a) A *read miss* to one in states $C(n)$, $D(n)$ or $S(n)$, results in a line in state $C(j)$ (for some j that may be 0) if main memory supplies it, or in state $S(j)$ ($j > 0$) if another cache does so. The remote operation is "main memory to cache" in the former case and "cache to cache" in the latter. A *read miss* access to a line in state $D(n)$ also causes a "cache to main memory" operation saving the old line. In this last case, the other n lines in the old sharing group change their state from $S(n)$ to $C(n-1)$, otherwise only the integer n in their state description changes to $n-1$.

(b) A *write* access to a line in any state results in a line in state $D(0)$. All other cache lines in the old sharing group (if the access is a *write hit*), or the new one (if a *write miss*), enter state I. The effect of a *write miss* on the old line and its sharing group is the same as that of a *read miss*. The bus operation is 'broadcast invalidate signal' if the access is a *write hit* in states other than $C(0)$ and $D(0)$; 'cache to cache' if *write miss* and the new line comes from another cache; 'main memory to cache' otherwise.

2.2.1.2 The Update Protocol

A cache line may be in one of the states $D(n)$, $S(n)$ and $C(n)$, whose definition is the same as in the previous subsection. The state I does not exist. The effect of a *read* access is the same as for the *Invalidate* protocol. A *write hit* access to a line in state $C(0)$ or $D(0)$ is local as before, and the resulting state is $D(0)$. All other *write* accesses are remote operations, the content of the new line being broadcast on the bus. The following state changes and additional bus operations take place:

(a) A *write hit* access causes a line to change its state from $C(n)$ or $S(n)$ to $D(n)$, and to remain in state $D(n)$ if it was there before. In all cases, the other n lines in the sharing group enter (or remain in) state $S(n)$. The "cache to cache" broadcast of the line is the only bus operation.

(b) A *write miss* access causes a line to join a new sharing group of some size j (possibly $j = 0$), and to enter state $D(j)$. The other j lines in that group enter state $S(j)$. If the state of the accessed line was $D(n)$, then the other n lines in the old sharing group change their state from $S(n)$ to $C(n-1)$, otherwise only the integer n in their state description changes to $n-1$. The bus operation consists of saving the old line ("cache to main memory"), if its state was $D(n)$; bringing the new line, part of which is to be overwritten ("main memory to cache" or "cache to cache", depending on the nature of the new sharing group); broadcasting the new line ("cache to cache").

2.2.2 Cache Line States

We assume that the K processors are statistically identical, and that their accesses are uniformly distributed over cache and main memory addresses. Those assumptions can be generalized, but at the expense of considerable increase in complexity. A processor 'computes' for an interval of time of average duration τ. At the end of that interval, which will be referred to as a 'think period', it accesses any one of its cache lines with probability $1/N$. If the access happens to be local, a new think period starts immediately; otherwise the processor joins the bus queue and remains passive until the required bus operation is complete.

Denote the steady-state probability that a given processor is 'thinking', by π. Then the average number of thinking processors is $\Upsilon = K\pi$. This quantity, which indicates the total rate at which useful work is being carried out, is called the *system power*; it is the performance measure of principal interest and the main object of the analysis.

An access to a cache line is a *read* with probability α, so a *write* occurs with probability $1-\alpha$. Also, and independently, a *hit* occurs with probability γ and a *miss* with probability $1-\gamma$. These probabilities are assumed fixed and known. Since m main memory lines are mapped onto one cache line, the uniform addressing assumption implies that a *miss* access will request any of the other $m-1$ lines with probability $1/(m-1)$.

Consider a particular line in one of the caches. Because of the symmetrical assumptions, we can concentrate on line 1 in cache 1. The state of this line at time t is assumed to be a Markov process. Transitions between states may occur when either processor 1 or one of the other processors make various kinds of accesses to line 1. The instantaneous rates of all these transitions have a common factor $\pi/(N\tau)$ (the rate at which a processor emerges from a think period and accesses line 1). Since the steady-state distribution of a Markov process does not change if all elements of its generator matrix are multiplied by the same number, we shall ignore this factor and include only those components of the transition rates which depend on the state. The two protocols have to be considered separately.

2.2.2.1 Equations for the Invalidate Protocol

Assuming the steady-state probabilities that the line is in state I, $C(n)$, $D(n)$ or $S(n)$ are p_I, $p_C(n)$, $p_D(n)$ and $p_S(n)$, respectively, let us consider state I. Any access by processor 1 to line 1, be it *read* or *write*, causes it to leave that state. On the other hand, it can enter state I from states $C(n)$, $D(n)$ or $S(n)$, $n > 0$, if :

(a) one of the other n processors in the sharing group makes a *write hit* access to line 1;

(b) one of the $K-n-1$ processors outside the sharing group is in state I and makes a *write* access to line 1 which hits the main memory address currently in line 1 of processor 1;

(c) one of the $K-n-1$ processors outside the sharing group is in a state other than I and makes a *write miss* access to line 1 which hits the main memory address currently in line 1 of processor 1.

Note that transitions (b) and (c) depend on the states of caches outside the group encompassed by the integer n. We approximate that dependency by introducing the steady-state probabilities into the transition rates. Moreover, the probability that a line is in a particular state, given that its cache is outside a sharing group of size $n+1$, is different from the corresponding unconditional probability. For example, such a line cannot be in state $S(j)$ for values of j exceeding $K-n-2$. To take account of this, the unconditional probabilities are multiplied by the 'renormalization factor', $a(n)$, given by:

$$a(n) = \left\{ p_I + \sum_{\ell=0}^{K-n-2} [p_C(\ell) + p_D(\ell) + p_S(\ell)] \right\}^{-1} . \tag{2.1}$$

Here and from now on, $p_S(0) = 0$ by definition.

As a consequence of the above approximations, the equation balancing the transitions into and out of state I is non-linear in the steady-state probabilities:

$$p_I = \sum_{n=0}^{K-1} [p_C(n) + p_D(n) + p_S(n)] \; \{ \, n(1-\alpha)\gamma + (K-n-1)(1-\alpha)a(n)$$

$$[p_I \frac{1}{m} + \sum_{j=0}^{K-n-2} (p_C(j) + p_D(j) + p_S(j))(1-\gamma)\frac{1}{m-1}] \, \} \; . \tag{2.2}$$

Analogous arguments lead to equations concerning states $C(n)$, $S(n)$ and $D(n)$. To write them, we need the probability, $q_C(i, j)$, that after leaving a sharing group of size $i + 1$ (as a result of a *miss* access), the line enters state $C(j)$. This is approximated as:

$$q_C(i, j) = \frac{p_C(j)}{\sum_{\ell=0}^{K-i-1}[p_C(\ell) + p_S(\ell)]} \quad , \quad j < K - i \; . \tag{2.3}$$

Similarly, the probability, $q_S(i, j)$, that after leaving a sharing group of size $i + 1$, the line enters state $S(j)$, is approximated as:

$$q_S(i, j) = \frac{p_S(j)}{\sum_{\ell=0}^{K-i-1}[p_C(\ell) + p_S(\ell)]} \quad , \quad j < K - i \; . \tag{2.4}$$

The balance equation for state $C(n)$ has the following form:

$$\begin{aligned}
& p_C(n) \, \{ \, (1-\alpha) + \alpha(1-\gamma) + n(1-\alpha)\gamma + n(1-\gamma) + (K-n-1)a(n) \\
& \quad [p_I \frac{1}{m} + \sum_{j=0}^{K-n-2} (p_C(j) + p_D(j) + p_S(j))(1-\gamma)\frac{1}{m-1}] \, \} \\
& = (n+1)(1-\gamma)p_C(n+1) + (1-\gamma)p_S(n+1) + p_I \alpha q_C(0, n) + \alpha(1-\gamma) \\
& \quad \sum_{i=0}^{K-1} [p_C(i) + p_D(i) + p_S(i)] q_C(i, n) + (K-n)\alpha a(n-1) \\
& p_C(n-1) \, \{ \, p_I \frac{1}{m} + \sum_{j=0}^{K-n-1} [p_C(j) + p_D(j) + p_S(j)](1-\gamma)\frac{1}{m-1}] \, \} \; .
\end{aligned} \tag{2.5}$$

The first term in the right-hand side of (2.5) reflects a reduction of the sharing group from $(n + 1)$ to n. The second corresponds to a transition from state $S(n + 1)$ to $C(n)$ when the line in the sharing group which is in state $D(n + 1)$ gets a *miss* access. The other terms are concerned with either the line in cache 1 moving out of its old sharing group into a new one, or another line joining the sharing group from outside. This equation holds for $n = 0, 1, \ldots K - 1$, with the understanding that $p_C(-1) = p_D(-1) = p_S(-1) = 0$.

The equation for state $S(n)$ is very similar:

$$\begin{aligned}
& p_S(n) \, \{ \, (1-\alpha) + \alpha(1-\gamma) + n(1-\alpha)\gamma + n(1-\gamma) + (K-n-1)a(n) \\
& \quad [p_I \frac{1}{m} + \sum_{j=0}^{K-n-2} (p_C(j) + p_D(j) + p_S(j))(1-\gamma)\frac{1}{m-1}] \, \} \\
& = n(1-\gamma)p_S(n+1) + p_I \alpha q_S(0, n) + \alpha(1-\gamma) \\
& \quad \sum_{i=0}^{K-1} [p_C(i) + p_D(i) + p_S(i)] q_S(i, n) + (K-n)\alpha a(n-1) \\
& p_S(n-1) \, \{ \, p_I \frac{1}{m} + \sum_{j=0}^{K-n-1} [p_C(j) + p_D(j) + p_S(j)](1-\gamma)\frac{1}{m-1}] \, \} \; .
\end{aligned} \tag{2.6}$$

In order to combine the states $D(n)$ ($n > 0$) and $D(0)$ into a single equation, let δ_B be the indicator function of condition B : it is 1 if B holds and 0 otherwise. Then we can write:

$$\begin{aligned}
& p_D(n) \, \{ \, (1-\alpha)\delta_{n>0} + \alpha(1-\gamma) + n(1-\alpha)\gamma + n(1-\gamma) + (K-n-1)a(n) \\
& \quad [p_I \frac{1}{m} + \sum_{j=0}^{K-n-2} (p_C(j) + p_D(j) + p_S(j))(1-\gamma)\frac{1}{m-1}] \, \}
\end{aligned}$$

$$= (n+1)(1-\gamma)p_D(n+1) + (1-\alpha)[1-p_D(0)]\delta_{n=0} + (K-n)\alpha a(n-1)$$

$$p_D(n-1)\ \{\ p_I\frac{1}{m} + \sum_{j=0}^{K-n-1}[p_C(j)+p_D(j)+p_S(j)](1-\gamma)\frac{1}{m-1}]\ \}\ . \tag{2.7}$$

Equations (2.2) - (2.7) can be easily rewritten in a way that expresses the vector of steady-state probabilities, $\mathbf{p} = (p_I; p_C(n), 0 \le n \le K-1; p_S(n), 1 \le n \le K-1; p_D(n), 0 \le n \le K-1)$, in terms of itself. That is, they can be written in the form

$$\mathbf{p} = f(\mathbf{p}) \tag{2.8}$$

Fixed-point equations of this type are normally solved iteratively: starting with an initial guess, $\mathbf{p}_0$, one computes successively $\mathbf{p}_{i+1} = f(\mathbf{p}_i)$, until two consecutive iterations are sufficiently close to each other. Of course, since $\mathbf{p}$ is a probability vector, its elements must be re-normalized at every iteration to ensure that they add up to 1.

2.2.2.2 Equations for the Update Protocol

Remember that state I does not exist in the *Update* protocol. The specification in subsection 2.2.1.2 leads to the following set of approximate equations.

State $C(n)$:

$$p_C(n)\ [\ (1-\alpha) + \alpha(1-\gamma) + n(1-\alpha)\gamma + n(1-\gamma) + (K-n-1)(1-\gamma)\frac{1}{m-1}\ ]$$
$$= (n+1)(1-\gamma)p_C(n+1) + (1-\gamma)p_S(n+1)$$
$$+(K-n)\alpha(1-\gamma)p_C(n-1)\frac{1}{m-1}$$
$$+\alpha(1-\gamma)\sum_{i=0}^{K-1}[p_C(i)+p_D(i)+p_S(i))]q_C(i,n)\ , \tag{2.9}$$

where $q_C(i,n)$ is defined as in the previous subsection.

State $S(n)$:

$$p_S(n)\ [\ (1-\alpha) + \alpha(1-\gamma) + n(1-\alpha)\gamma + n(1-\gamma) + (K-n-1)(1-\gamma)\frac{1}{m-1}\ ]$$
$$= n(1-\gamma)p_S(n+1) + n(1-\alpha)\gamma[p_C(n)+p_D(n)]$$
$$+(K-n)(1-\gamma)\frac{1}{m-1}\ \{\ p_S(n-1) + (1-\alpha)[p_C(n-1)+p_D(n-1)]\ \}$$
$$+\alpha(1-\gamma)\sum_{i=0}^{K-1}[p_C(i)+p_D(i)+p_S(i))]q_S(i,n)\ . \tag{2.10}$$

Again, $q_S(i,n)$ is defined as in the previous subsection.

State $D(n)$:

$$p_D(n)\ [\ (1-\gamma) + n(1-\alpha)\gamma + n(1-\gamma) + (K-n-1)(1-\gamma)\frac{1}{m-1}\ ]$$
$$= (n+1)(1-\gamma)p_D(n+1) + (1-\alpha)\gamma[p_C(n)+p_S(n)] + (K-n)\alpha(1-\gamma)\frac{p_D(n-1)}{m-1}$$
$$+(1-\alpha)(1-\gamma)\sum_{i=0}^{K-1}[p_C(i)+p_D(i)+p_S(i))]q_D(i,n)\ , \tag{2.11}$$

where $q_D(i,j)$ is defined as:

$$q_D(i,j) = \frac{p_D(j)}{\sum_{\ell=0}^{K-i-1}p_D(\ell)}\quad,\quad j < K-i\ .$$

Once more, the vector of unknown probabilities can be expressed, through equations (2.9) - (2.11), in terms of itself. The fixed-point problem is solved iteratively, starting with an initial guess and re-normalizing at every iteration.

2.2.3 Bus Queue Metrics

The bus can be modelled as a single server FIFO queue which is fed with requests by K finite sources (Figure 2.3).

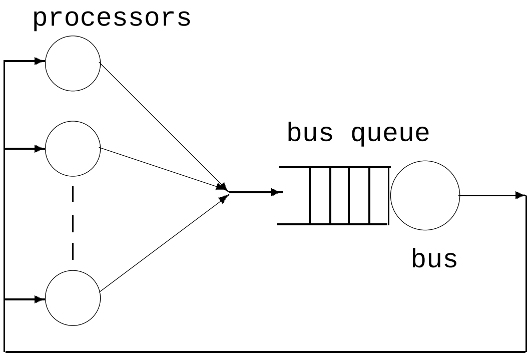


Figure 2.3: The cache queue

If a processor is in 'think' state, then the rate, λ, at which it makes a request for bus service is equal to the rate at which the think period is completed, multiplied by the probability that the resulting cache access is not local. In the case of the *Invalidate* protocol, this gives:

$$\lambda = \frac{1}{\tau}\,\{\,1 - \;\alpha\gamma(1-p_I) - (1-\alpha)\gamma[p_C(0)+p_D(0)]\,\}\;.$$

For the *Update* protocol, that rate is equal to:

$$\lambda = \frac{1}{\tau}\,\{\,1 - \;\alpha\gamma - (1-\alpha)\gamma[p_C(0)+p_D(0)]\,\}\;.$$

Bus service times may have different averages, depending on the protocol and on the kind of operation that is carried out. These differences could be taken into account by introducing multiple request types. However, the resulting model would not have a product-form solution and its number of states would rise very quickly with K. An acceptable approximation is obtained by using a common average service time, taken as a weighted mean of the possible bus operation times. Thus, if a bus operation of average length b_j is requested with probability r_j, for $j = 1, 2, \ldots, J$, then the the overall average bus service time, b, is equal to:

$$b = \sum_{j=1}^{J} r_j b_j\;.$$

Clearly, in order to determine the probabilities r_j, it is necessary first to solve the appropriate cache line model. The weighted averages for the *Invalidate* and *Update* protocols are evaluated below.

2.2.3.1 Average Bus Service Time for Invalidate Protocol

Denote by τ_1, τ_2 and τ_3 the average lengths of the 'cache to cache', 'cache to main memory' (or 'main memory to cache') and 'invalidate signal' operations, respectively. Then under the *Invalidate* protocol, there are 5 types of bus requests, with service times $b_1 = \tau_1$, $b_2 = \tau_2$, $b_3 = \tau_1 + \tau_2$, $b_4 = 2\tau_2$ and $b_5 = \tau_3$. These occur in the following circumstances and with the following probabilities:

Type 1 : A *miss* access in state $C(n)$ or $S(n)$ or any access in state I, after which the line joins a sharing group where one of the lines is in state $D(j)$. Let $d(n)$ be the probability that a line is in state $D(j)$, given that it is outside a sharing group of size $n+1$:

$$d(n) = \frac{\sum_{i=0}^{K-n-2} p_D(i)}{\sum_{i=0}^{K-n-2}[p_C(i)+p_S(i)+p_D(i)] + p_I}\;.$$

Then we have

$$r_1 = \sum_{n=0}^{K-1} [p_C(n) + p_S(n)](1-\gamma)(K-n-1)\frac{d(n)}{m-1} + p_I(K-1)\frac{d(0)}{m-1} \ .$$

Type 2 : A *miss* access in state $C(n)$ or $S(n)$ or any access in state I, after which the line joins a sharing group where there is no line in state $D(j)$.

$$r_2 = \sum_{n=0}^{K-1} [p_C(n) + p_S(n)](1-\gamma)[1-(K-n-1)\frac{d(n)}{m-1}]$$

$$+p_I[1-(K-1)\frac{d(0)}{m-1}] \ .$$

Type 3 : A *miss* access in state $D(n)$, after which the line joins a sharing group where one of the lines is in state $D(j)$.

$$r_3 = \sum_{n=0}^{K-1} p_D(n)(1-\gamma)(K-n-1)\frac{d(n)}{m-1} \ .$$

Type 4 : A *miss* access in state $D(n)$, after which the line joins a sharing group where there is no line in state $D(j)$.

$$r_4 = \sum_{n=0}^{K-1} p_D(n)(1-\gamma)[1-(K-n-1)\frac{d(n)}{m-1}] \ .$$

Type 5 : A *write hit* access in any state other than I, $C(0)$ and $D(0)$.

$$r_5 = [1 - p_I - p_C(0) - p_D(0)](1-\alpha)\gamma \ .$$

2.2.3.2 Average Bus Service Time for Update Protocol

Under the *Update* protocol, there are 7 types of bus requests, with service times $b_1 = \tau_1$, $b_2 = \tau_2$, $b_3 = \tau_1 + \tau_2$, $b_4 = 2\tau_1$, $b_5 = 2\tau_2$, $b_6 = 2\tau_1 + \tau_2$ and $b_7 = 2\tau_2 + \tau_1$. The circumstances in which these occur, and their probabilities, are as follows:

Type 1 : A *write hit* access in any state other than $C(0)$ and $D(0)$, or a *read miss* access in states $C(n)$ or $S(n)$, after which the line joins a sharing group where one of the lines is in state $D(j)$.

$$r_1 = (1-\alpha)\gamma[1 - p_C(0) - p_D(0)] + \sum_{n=0}^{K-1} [p_C(n) + p_S(n)]\alpha(1-\gamma)(K-n-1)\frac{d(n)}{m-1} \ .$$

Type 2 : A *read miss* access in states $C(n)$ or $S(n)$, after which the line joins a sharing group where there is no line in state $D(j)$.

$$r_2 = \sum_{n=0}^{K-1} [p_C(n) + p_S(n)]\alpha(1-\gamma)[1-(K-n-1)\frac{d(n)}{m-1}] \ .$$

Type 3 : A *read miss* access in state $D(n)$, after which the line joins a sharing group where one of the lines is in state $D(j)$, or a *write miss* access in states $C(n)$ or $S(n)$, after which the line joins a sharing group where there is no line in state $D(j)$.

$$r_3 = \sum_{n=0}^{K-1} p_D(n)\alpha(1-\gamma)(K-n-1)\frac{d(n)}{m-1}$$

$$+ \sum_{n=0}^{K-1} [p_C(n) + p_S(n)](1-\alpha)(1-\gamma)[1-(K-n-1)\frac{d(n)}{m-1}] \ .$$

Type 4 : A *write miss* access in states $C(n)$ or $S(n)$, after which the line joins a sharing group where one of the lines is in state $D(j)$.

$$r_4 = \sum_{n=0}^{K-1} [p_C(n) + p_S(n)](1-\alpha)(1-\gamma)(K-n-1)\frac{d(n)}{m-1} \ .$$

Type 5 : A *read miss* access in state $D(n)$, after which the line joins a sharing group where there is no line in state $D(j)$.

$$r_5 = \sum_{n=0}^{K-1} p_D(n)\alpha(1-\gamma)[1-(K-n-1)\frac{d(n)}{m-1}] \ .$$

Type 6 : A *write miss* access in state $D(n)$, after which the line joins a sharing group where one of the lines is in state $D(j)$.

$$r_6 = \sum_{n=0}^{K-1} p_D(n)(1-\alpha)(1-\gamma)(K-n-1)\frac{d(n)}{m-1} \ .$$

Type 7 : A *write miss* access in state $D(n)$, after which the line joins a sharing group where there is no line in state $D(j)$.

$$r_7 = \sum_{n=0}^{K-1} p_D(n)(1-\alpha)(1-\gamma)[1-(K-n-1)\frac{d(n)}{m-1}] \ .$$

2.2.4 Performance Metrics

Assuming that the service times and think periods are exponentially distributed, the steady-state probability , p_i, that there are i requests in the bus queue is equal to (e.g., see [Mitrani 87]):

$$p_i = \frac{K!}{(K-i)!}(\lambda b)^i p_0 \ , \tag{2.12}$$

where:

$$p_0 = \left[\sum_{i=0}^{K} \frac{K!}{(K-i)!}(\lambda b)^i\right]^{-1} \ .$$

The following performance measures are then obtained [Mitrani 87]:

Bus utilisation U :

$$U = 1 - p_0 \ .$$

Average response time W **for a request** :

$$W = \frac{Kb}{U} - \frac{1}{\lambda} \ .$$

Probability π **that a given processor is thinking** :

$$\pi = \frac{\frac{1}{\lambda}}{\frac{1}{\lambda} + W} = \frac{U}{Kb\lambda} \ .$$

System power Υ :

$$\Upsilon = K\pi = \frac{U}{b\lambda} \ .$$

2.2.5 Comparison to Numerical Simulation Results

Several numerical experiments have been carried out, where the performance of the *Invalidate* and *Update* protocols was evaluated as described in the previous sections. The ratio of main memory size to cache size was taken as $m = 11$, and the average think period, τ, was 1. In the first, and larger group of results, the following average bus transfer times were used: ‘cache to cache’=1/6; ‘cache to main memory’=‘main memory to cache’=1/4; ‘invalidate signal’=1/54.

Figure 2.4 shows how the performance of the the two protocols, measured by the system power, varies with the number of processors. The fraction of *read* accesses, and the fraction of *hit* accesses, are quite high, at 0.85 and 0.8, respectively. In this case, the *Update* protocol gives better throughout. Moreover, the *Invalidate* protocol displays clear symptoms of ‘thrashing’, whereby having reached a maximum, the power begins to deteriorate if the number of processors continues to increase.

The figure also includes performance curves obtained by simulation, for both exponentially distributed (with the above mean values) and constant bus transfer times. The analytical approximations and the simulations are very close.

For Figure 2.5, the number of processors and the *hit* probability were fixed, while the *read* probability was varied between 0 and 1. It can be seen that when most accesses are *write*, the *Invalidate* protocol is better (because it takes less time to send an *invalidate* signal than a whole line), whereas the situation is reversed when most accesses are *read* (because there are fewer transfers between cache and main memory under the *Update* protocol).

Figure 2.6 shows that the effect of fixing the probability of *read* and varying that of *hit* is similar. When the *hit* probability is low there is no advantage in broadcasting the new content of changed lines, only the penalty of longer bus operations.

Figure 2.7 is of the same type as Figure 2.6, but deals with a system where the bus operations are much longer relative to the processor think periods: ‘cache to cache’=6; ‘cache to main memory’=‘main memory to cache’=8; ‘invalidate signal’=2. Now the *hit* probability needs to be almost 1 before the system power rises above 2. The processors spend most of their time waiting for the bus, and because the *invalidate* signal is faster, that protocol is slightly better.

These figures show that rather complex cache coherence protocols can be analysed approximately and their performance can be evaluated with acceptable accuracy. Such approximations are certainly worthwhile, because a numerical solution, even one involving fixed-point iterations, is several orders of magnitude faster than a detailed simulation.

The methods described above can be applied to different protocols and different computer architectures, e.g., two multiprocessing nodes connected to a single shared memory as in Figure 2.8, a likely scenario in a fault tolerant system (and a minimalist example of a distributed shared memory system composed from more than one multiprocessor). Moreover, these models can be extended to cover other aspects of system behaviour, in addition to cache coherence.

The difficulty with this analytical approach to modelling is that if the mathematics is not appropriate to a new architecture, or even a new variant of an architecture already being studied, then a new mathematical formulation may be needed, and this is not conducive to the rapid evaluation of a number of disparate alternatives within the design process. This has led most designers to more direct numerical simulation methods using commercially available software tools based on queueing models. It is to this approach we now turn.

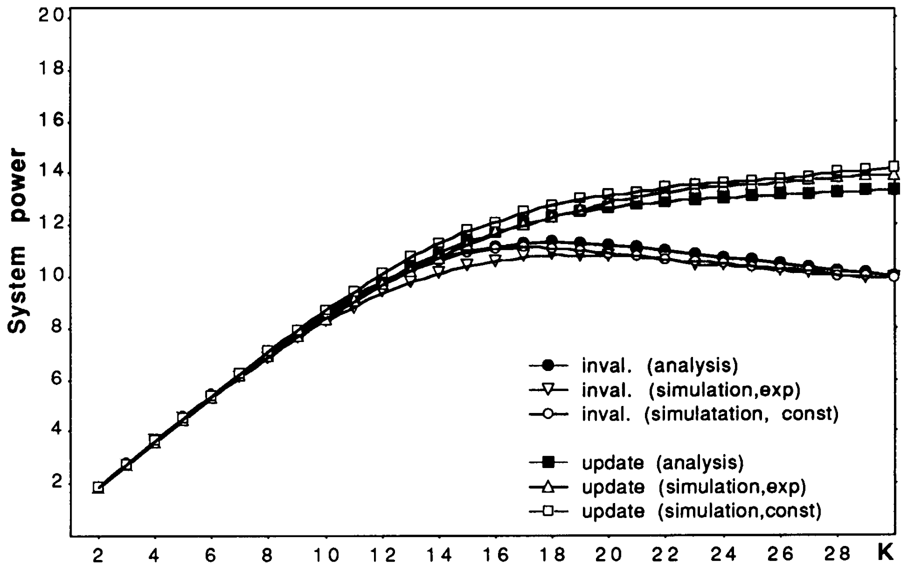


Figure 2.4: System power versus number of processors, α = 0.85, γ = 0.8

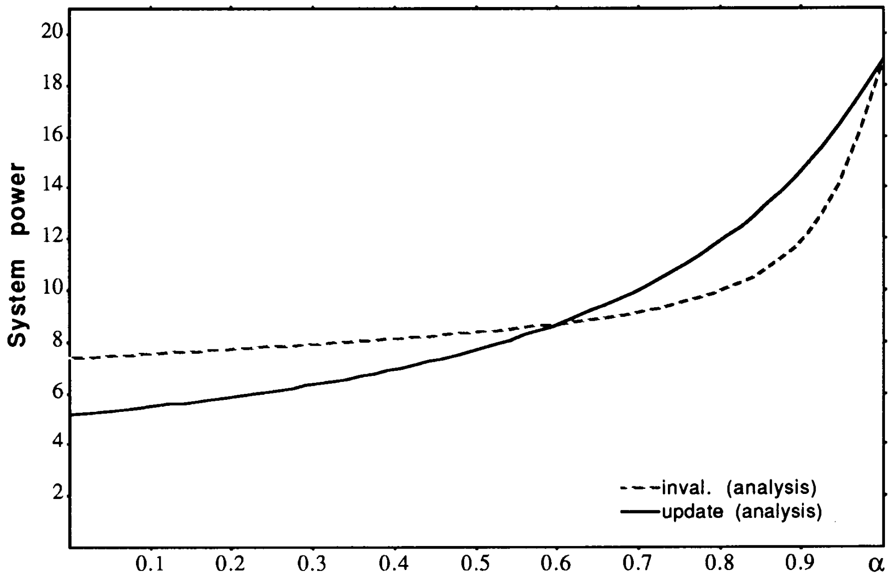


Figure 2.5: System power versus *read* probability, K = 25, γ = 0.8

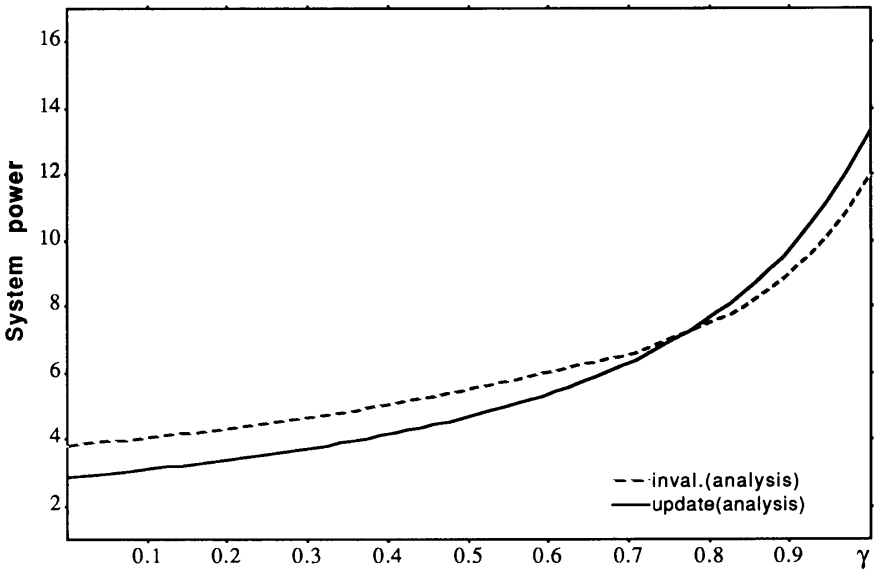


Figure 2.6: System power versus *hit* probability, K = 25, α = 0.5

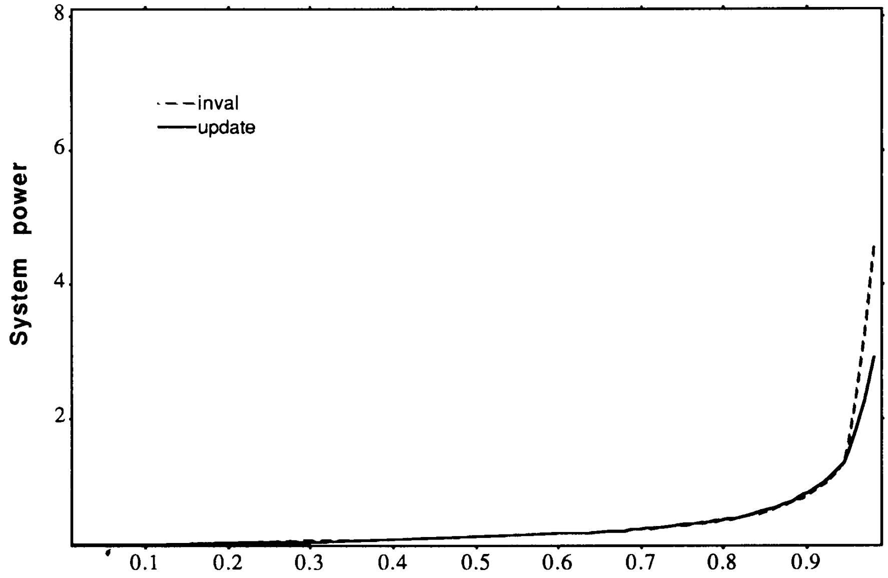
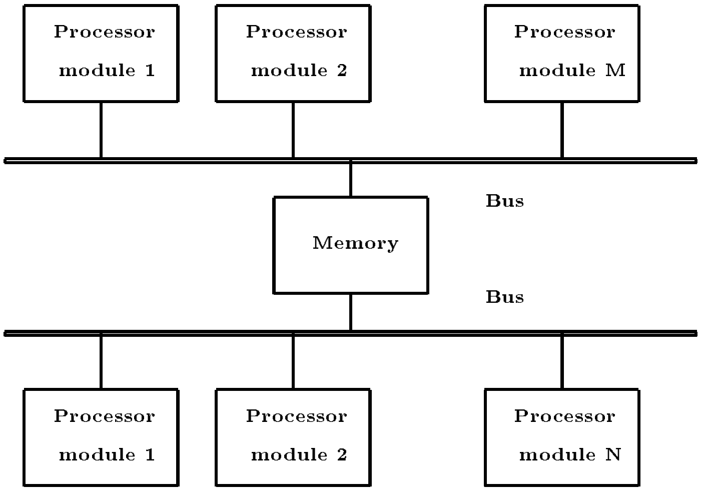


Figure 2.7: System power versus *hit* probability, K = 8, α = 0.95

.



.

Figure 2.8: Two multiprocessing nodes connected to a single shared memory

2.3 Queueing Models that include Checkpointing and Recovery[3]

The great virtue of numerical simulation is that it is supported by a wide range of computer software, such as the QNAP2 and SMPL packages. This immediately allows more complex structures to be evaluated. We now present extended cache coherent queueing models that incorporate checkpointing and recovery operations.

Checkpointing and recovery involve the caches, so the system behaviour must still be studied at the lowest level. The basic component of the workload is still the cache line, while the basic components of the simulation model remain the processors, the caches, the cache controllers, the bus and the shared memory. The processor module will generally incorporate an internal or primary cache and an external or secondary cache, as in Figure 2.9, but we may simplify this model by assuming that the internal pipelined structure of modern RISC processors allows the access to the internal cache to proceed in parallel with the instruction execution, yielding the processor model shown in Figure 2.10.

For comparison, five different types of multiprocessing systems have been simulated, three with different checkpointing and recovery mechanisms, plus two that do not include checkpointing and recovery. The multiprocessors that include checkpointing and recovery are a CARER system [Wu et al 90, Ahmed et al 90], a system using the FASST Recovery Protocol as described in Chapter 1 [Morin et al 92], and a system that uses the SMRC protocol [Ors et al 94a, Ors et al 94b]. The latter two assume specialized recoverable shared memory (*stable memory*). The two systems without checkpointing and recovery differ by their use of the *Dragon* and *Berkeley* cache coherence protocols respectively.

The three recoverable architectures are represented by the queueing model of Figure 2.11. The stations represent the processors (P_n), the external caches (CH_n), and the bus. The recoverable shared memory is modelled by three stations. The model includes some sources to simulate unrecoverable operations (interrupts and I/O's) and rollbacks. The sources connected to the processors simulate the establishment of a checkpoint due to an atomic operation, which is generally of two types:

Explicit Recovery Points : The main source of these checkpoints are interrupts and I/O operations.

Implicit Recovery Points : These arise from to the special characteristics of the recovery protocol.

[3] The following sections contributed by R. Ors, J.J. Serrano, V. Santonja and P. Gil, Universidad Politécnica de Valencia, and A. Pérez and S. Rodríguez, Universidad Politécnica de Madrid

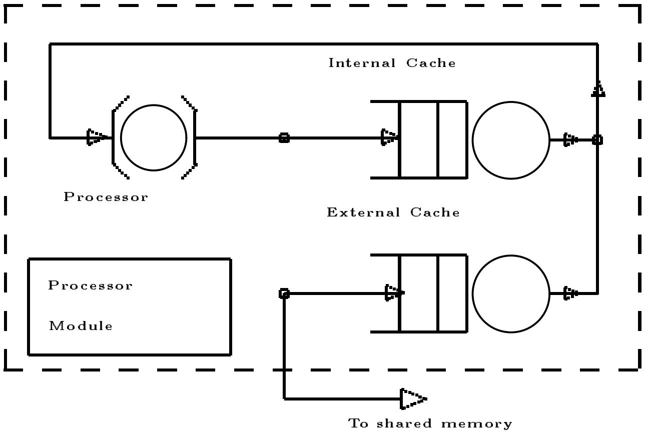


Figure 2.9: Processor Model

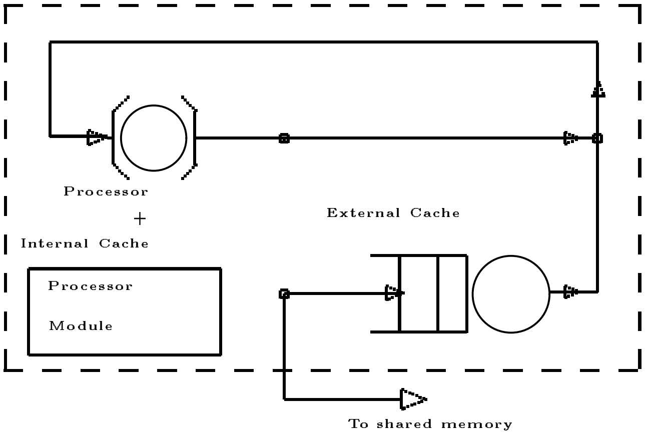


Figure 2.10: Simplified Processor Model

These models have been simulated firstly with QNAP2 and later with SMPL [MacDougall 89], which is three times faster than QNAP2. A Smalltalk version has also been developed. In all cases the the mean number of active processors in the system, the *Processing Power*, is used as a performance index, since it is representative for a comparative analysis. In general the analyses assume the following input parameters:

Cache size (lines) :	128 (L)
Program size (lines) :	5120 (40L)
Shared memory size (lines) :	51200 (400L)
Number of Processors :	20
Probability of repeating a line :	0.5
Read probability :	0.8
Relation table size :	2L
Threshold for dirty lines :	0.75L
Atomic operations rate :	10^{-5}
Workload parameter A :	5
Locality parameter θ :	2.5

The performance analysis is highly dependent on the workload used, so the workload needs to be much more carefully specified. Four kinds of test workloads have been used to analyse these models, one called WL-I that is based on a probabilistic model, one fixed workload or natural trace, called WL-II, for debugging and testing, and another two, called WL-III and WL-IV, based on a synthetic trace. In the first, WL-I, the workload is characterized by the following parameters:

(a) The probability P_{rep} of making an access to the same block,

(b) The probability P_{sh} of making an access to a shared block, and

(c) A uniform distribution in the interval $(0, S_{dh})$ for shared data and (S_{hd}, NS_{dh}) for private data.

The latter workloads, WL-III and WL-IV, are taken from [Thiebaut et al 92], adapted to the models under consideration. WL-IV is used when the number of shared lines is limited, otherwise WL-III is used. These workloads simulate, based on a hyperbolic distribution, the jumps made by the program counter during the execution of a program, and are parameterized by:

(a) Cache capacity,

(b) A constant that characterizes the load, and

(c) A parameter that characterizes the spatial locality and the memory size.

Since these are simulations, it is relatively easy to examine the sensitivity of the results to the various workload parameters. In Figure 2.12 we can see the influence of cache size on the performance of the three checkpointing and recovery systems and the multiprocessors. The performance is obviously sensitive to the cache size and the influence is similar for all the systems.

Figure 2.13 illustrates the influence of the program size on the performance of the different systems. When the program reaches 16 times the cache line size, the performance does not change anymore. The variability is similar for all the systems.

Figure 2.14 shows the influence of the read probability on the performance of the different systems. As reads produce fewer dependencies than writes, the best case arises when we do not have writes, i.e. for $P_r = 1.0$. We can see that in all the systems this parameter has a significant influence on the system performance.

Figure 2.15 shows the influence of the checkpointing rate on the performance of the different systems. Obviously CARER is the better at high checkpointing rates.

2.4 Dependability analysis

Performance is an important issue for any computing system, but at the heart of a fault-tolerant computer system must be its ability to survive when a non-fault-tolerant system would fail. This gives rise to notions of dependability, reliability, availability and performability (see [Lee et al 90]). Reliability and availability can be analysed using Markov chain models. Performability can be analysed using Reward Markov models, where the reward is the *Processing Power*, i.e. the mean number of active processors in the configuration considered. Each configuration has a queue equivalent model with the number of processors that are not in a crash state. The parameters used in the reliability, availability and performability models are:

P_p	Permanent fault rate
f_t	Transient fault rate
λ_e	Fault rate for a processor module
λ_b	Fault rate for the bus
λ_m	Fault rate for the shared memory
δ	Repair rate
C	Transient fault coverage
D	Permanent fault coverage
E	Shared memory fault coverage
$1/\alpha$	Average time between two recovery points
$1/\beta$	Average time for a recovery point
$1/\rho$	Average time for a rollback
N	Number of processor

2.4.1 Multiprocessor system without checkpointing and recovery

By way of reference, let us first consider a multiprocessor without checkpointing and recovery, so that the effects of introducing these features can be clearly seen. Figure 2.16 shows the time evolution of the reliability $R(t)$ for such a system. We can see two sets of lines on the graph, corresponding to two fault rates λ_m for the shared memory. The influence of the fault rate λ_e for processor modules can be more clearly observed for one of these sets than the other.

In Figure 2.17 we can see the availability $A(N)$ versus the number of processors N, using the fault rate λ_m for the shared memory and the fault rate λ_e for processor modules as parameters. The influence of these parameters on the availability is similar to their influence on the reliability.

In Figure 2.18 we can see the time evolution of the performability $P(t)$ of the system. We can observe that this evolution is similar to the time evolution of the reliability $R(t)$ shown in Figure 2.16. Figure 2.19 shows the influence of the number of processors N on the time evolution of performability $P(t)$ for such a system.

2.4.2 Non-degradable multiprocessor system with checkpointing and recovery

Now let us consider a CARER system, which is not *degradable*, in the sense that it cannot survive permanent processor faults. This will allow us (later) to more easily assess the effects of introducing degradability. Figure 2.20 shows the time evolution of the reliability $R(t)$ for such a system, while in Figure 2.21 we can see the availability $A(\delta)$ against the repair rate δ.

Figures 2.22, 2.23, 2.24 and 2.25 show the time evolution of the performability $P(t)$. In Figure 2.22 the processor and memory fault rates λ_e and λ_m are used as parameters, in Figure 2.23 the transient fault coverage C is a parameter, in Figure 2.24 the permanent fault rate P_p is a parameter, and finally in Figure 2.25, we can see the influence of the number of processors N on the performability.

2.4.3 Degradable multiprocessor system with checkpointing and recovery

Finally, let us now consider a degradable multiprocessor system (i.e. one that can survive permanent processor faults) with checkpointing and recovery, such as is proposed for FASST. Figure 2.26 shows the time evolution of the reliability $R(t)$ for such a system, parameterized by the transient fault coverage C.

Figure 2.27 shows the availability $A(\delta)$ for such a system versus the repair rate δ. We can see two set of lines, one for each memory fault rate λ_m, indicating the influence of this parameter on the availability; in each set the lines are very close, indicating that the processor fault rate λ_e has very little influence.

Figure 2.28,2.29,2.30,2.31 and 2.32 show the temporal evolution of the performability $P(t)$. In Figure 2.28 the processor and memory fault rates λ_e and λ_m are used as parameters. In Figure 2.29 the transient fault coverage C is a parameter, in Figure 2.30 the permanent fault coverage D is a parameter, in Figure 2.31 the permanent fault rate P_p is a parameter, and finally in Figure 2.32 we can see the influence of the number of processors N.

2.5 Summary

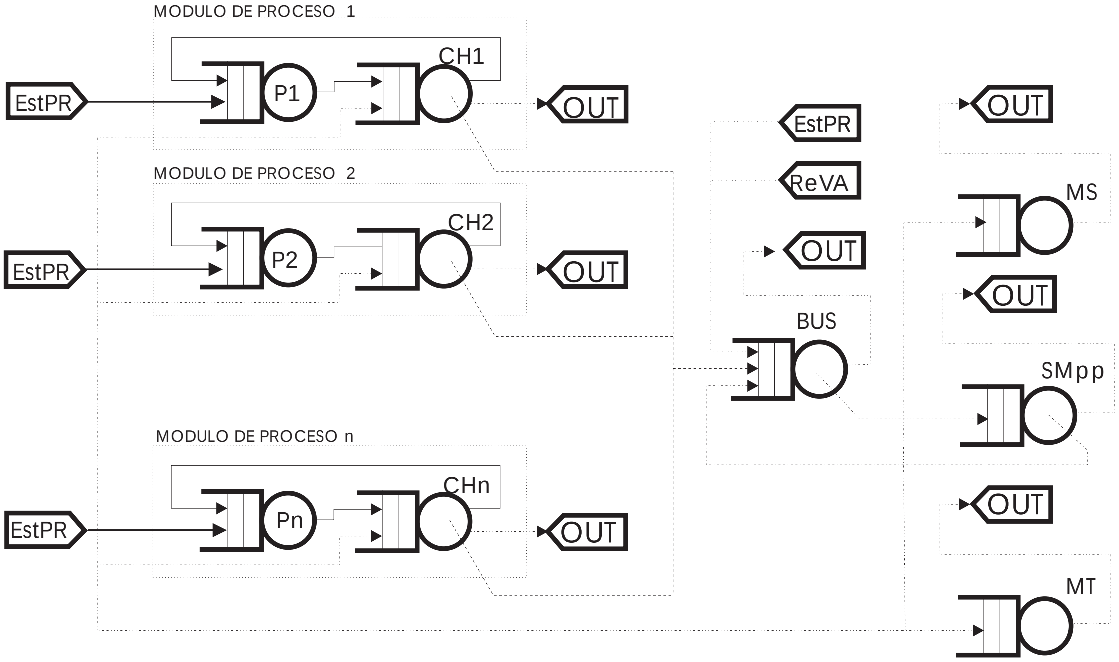
In Figures 2.33 and 2.34, we can see the temporal evolution of the system performability. As one might expect, a basic multiprocessor system without checkpointing and recovery initially has the best performability, but quickly, due to its low reliability, becomes the worst system. Systems with checkpointing and recovery but no degradation again have good initial performance, but since they do not tolerate permanent faults, their reliability decreases and their performability is affected as a result. Good long term performability is offered only by systems that tolerate permanent and transient faults, like those that use the FASST or SMRC recovery protocol. The FASST and SMRC protocols have the same functionality; the improved performance of the SMRC protocol is due to migration of most of the algorithms into the recoverable shared memory, which then becomes a system bus master to establish a recovery point by using the cache coherency mechanisms to broadcall state from the processors.

Bear in mind that these performability results are for a single cache coherent multiprocessing node. The situation is a little more complex with more than one multiprocessing node, in that a mechanism (let us call it a *bridge*) must be constructed to carry coherence traffic between the busses of the nodes, since although there is more than one bus, there is only one set of common data. The *bridge* function can be modelled probabilistically or with a boundary analysis that considers only the upper and lower bounds of the system performance [Yang et al 88]; let us just briefly consider this last option for a minimalist dual-node configuration.

The upper bound is obtained by assuming that the two busses don't have any shared data, i.e. the bridge is never used, so that there are, in effect, two independent busses (see Figure 2.36). For the lower bound we assume that all the data is shared, so the bridge is always used, and effectively the system has only one bus (see Figure 2.37). The sources and sinks in these models represent the extra load due to the cache coherence protocol.

These models have been studied with *Write-Once* [Goodman 84], *Write-through* [Fielland et al 84], *Berkeley* [Katz et al 85], *Synapse* [Frank 84], *Illinois* [Patel et al 84], *Firefly* [Thacker et al 87], and *Dragon* [McCreight 84] cache coherence protocols. As an example, the results for the *Write-Once* protocol (using the workload parameters $fr = 0.7$, $fw = 0.3$, $h = 0.9$, $md = 0.2$, $umd = 0.1$, $Nsb = 32$, $qs = 0.2$, $Sc = 2K$, $Z = 2$, $tcc = 1$, $tcb = 1$ and $tcm = 4$) are shown in the Figure 2.35.

As we shall see in the next chapter, however, the FASST architecture is defined for just a single multiprocessing node; extension to more than one such node is really beyond the scope of this book.



.

Figure 2.11: FASST queueing model

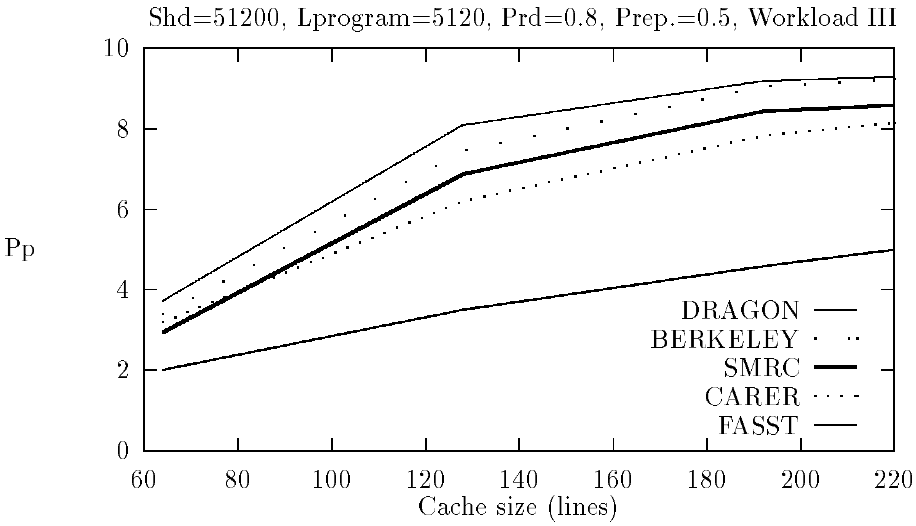


Figure 2.12: Influence of cache size on performance

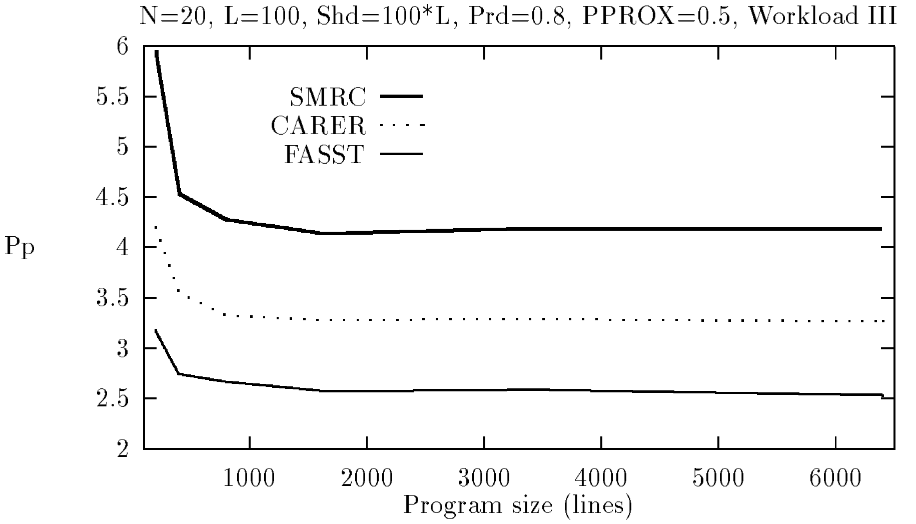


Figure 2.13: Influence of program size on performance

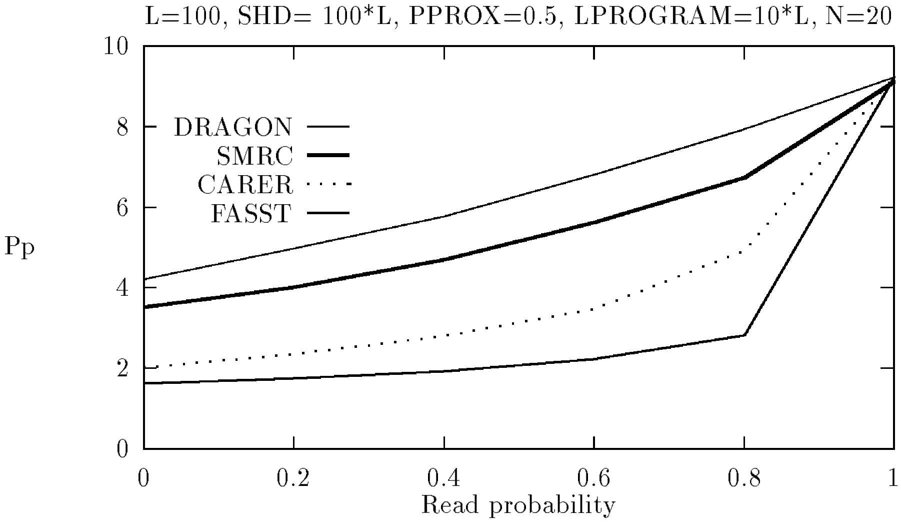


Figure 2.14: Influence of read probability on performance

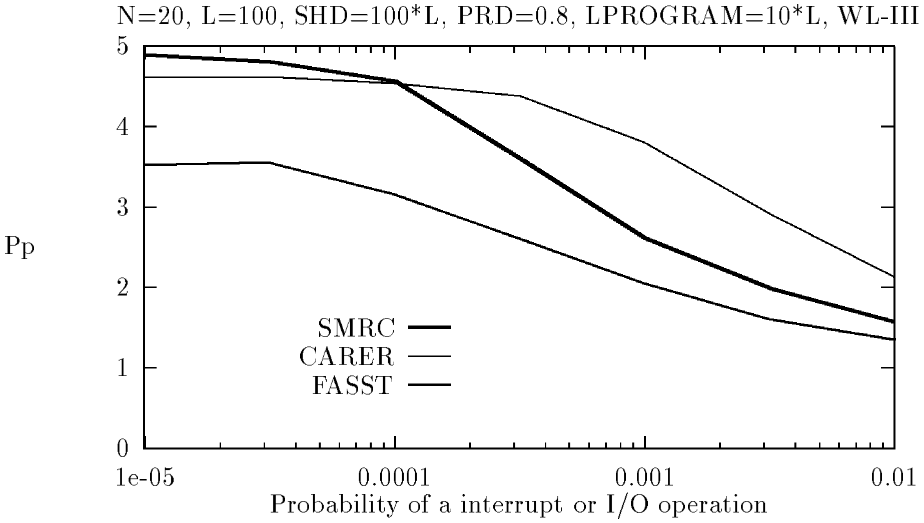


Figure 2.15: Influence of the checkpointing rate on performance

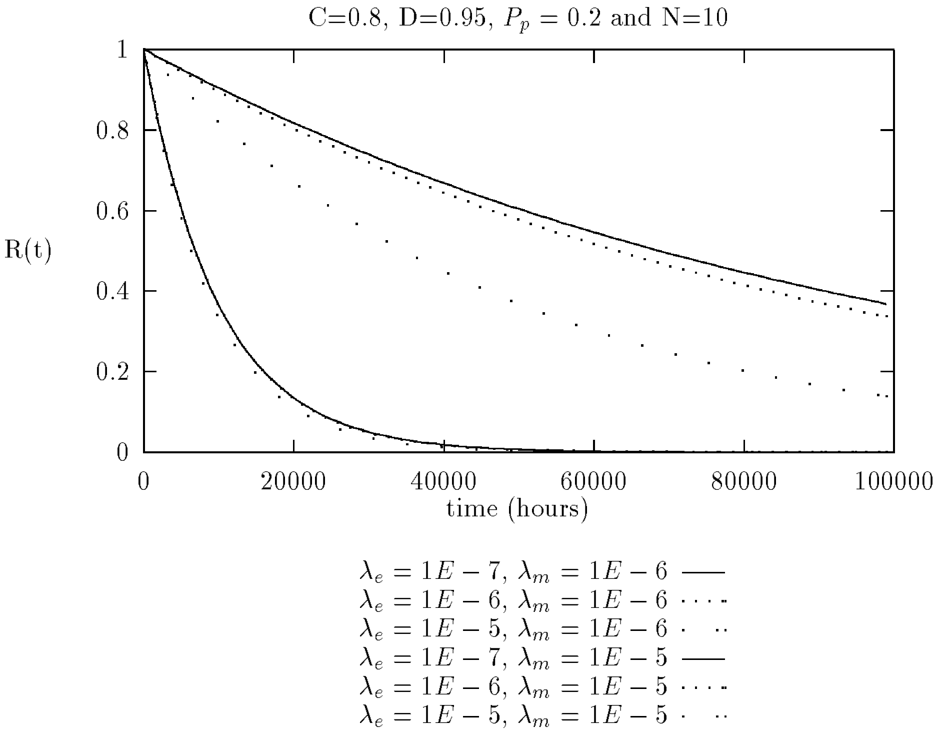


Figure 2.16: Reliability for a multiprocessor system without checkpointing and recovery

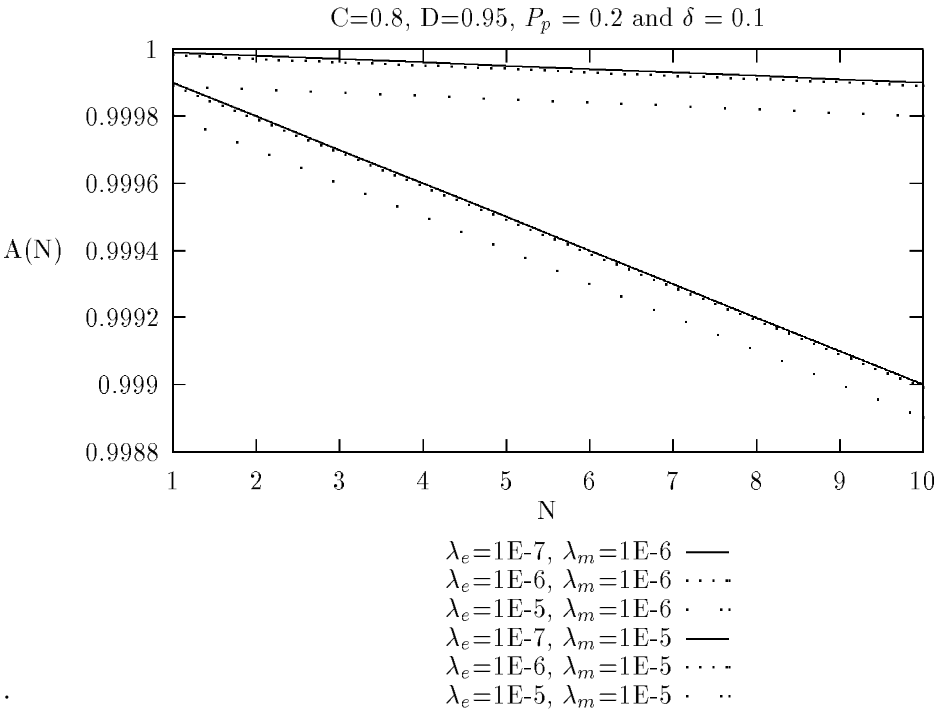
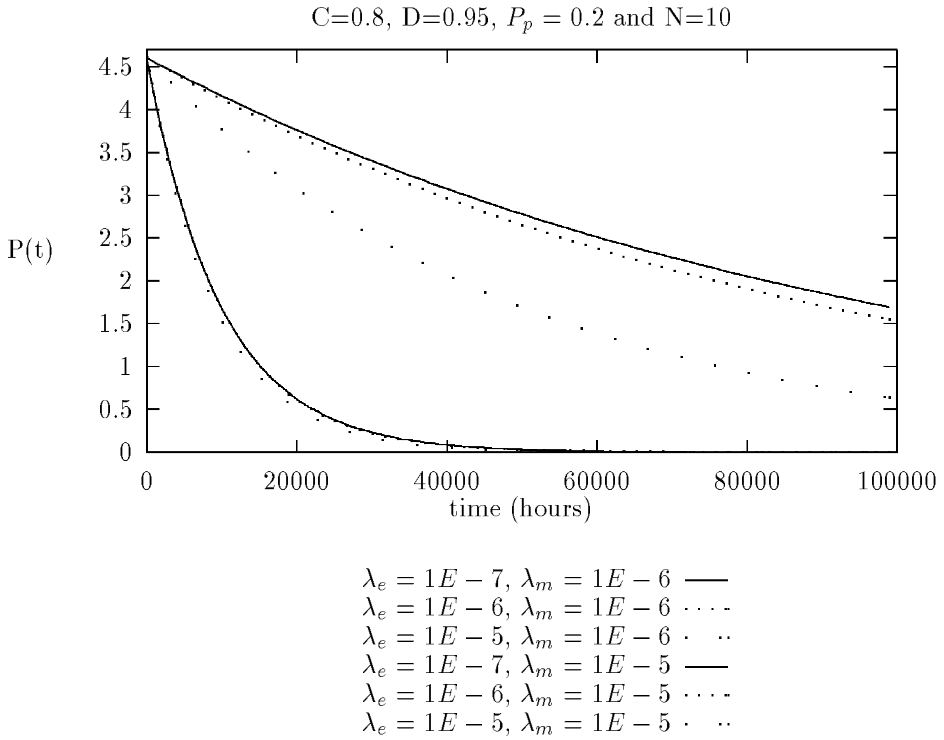


Figure 2.17: Availability for a multiprocessor system without checkpointing and recovery



.

Figure 2.18: Influence of the processor and memory fault rates on the performability of a multiprocessor system without checkpointing and recovery

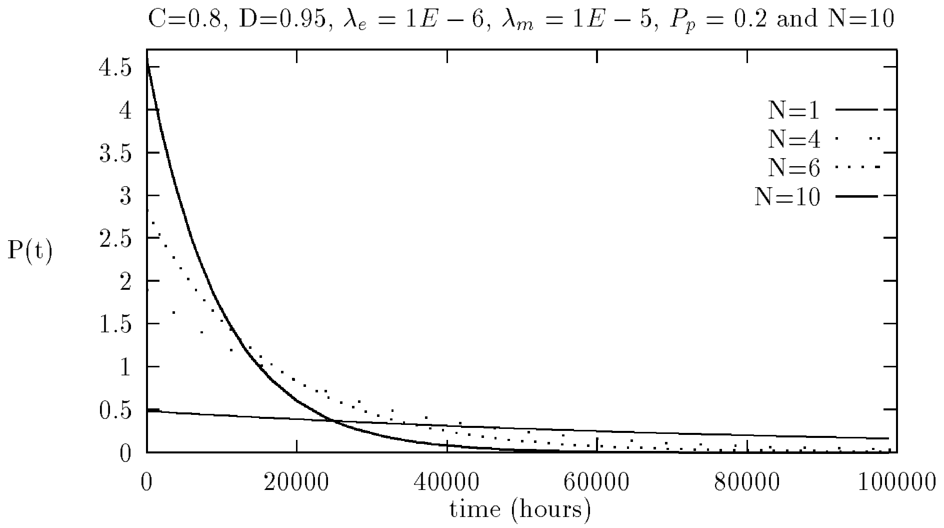


Figure 2.19: Influence of the number of processors on the performability of a multiprocessor system without checkpointing and recovery

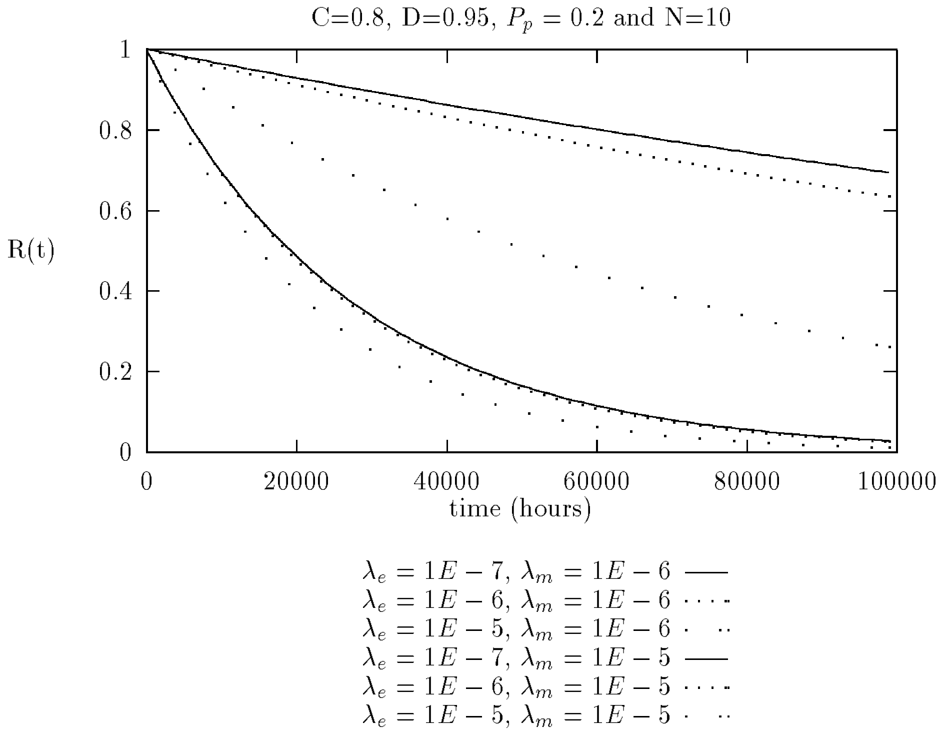


Figure 2.20: Reliability for a non-degradable (CARER) multiprocessor system with checkpointing and recovery

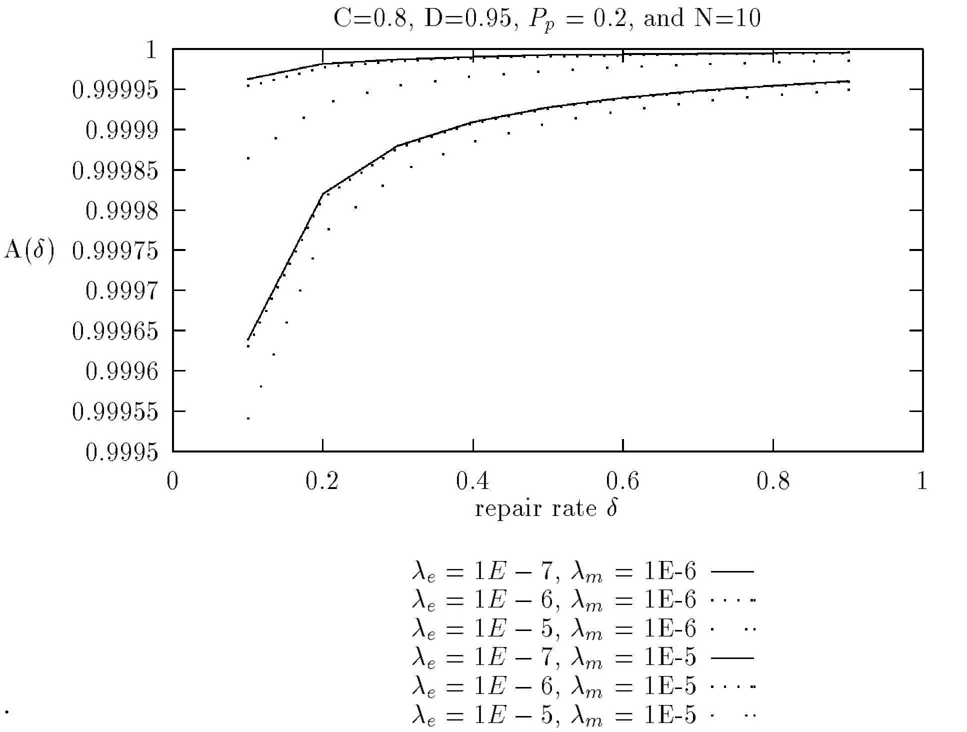
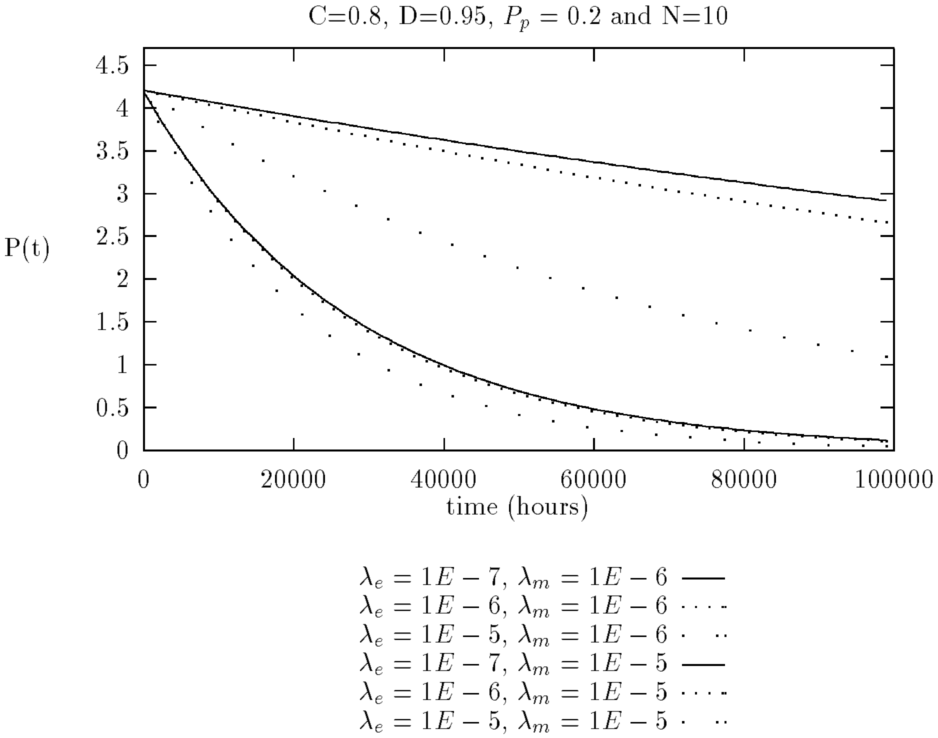


Figure 2.21: Availability for a non-degradable (CARER) multiprocessor system with checkpointing and recovery



.

Figure 2.22: Influence of processor and memory fault rates on the performability of a non-degradable (CARER) multiprocessor system with checkpointing and recovery

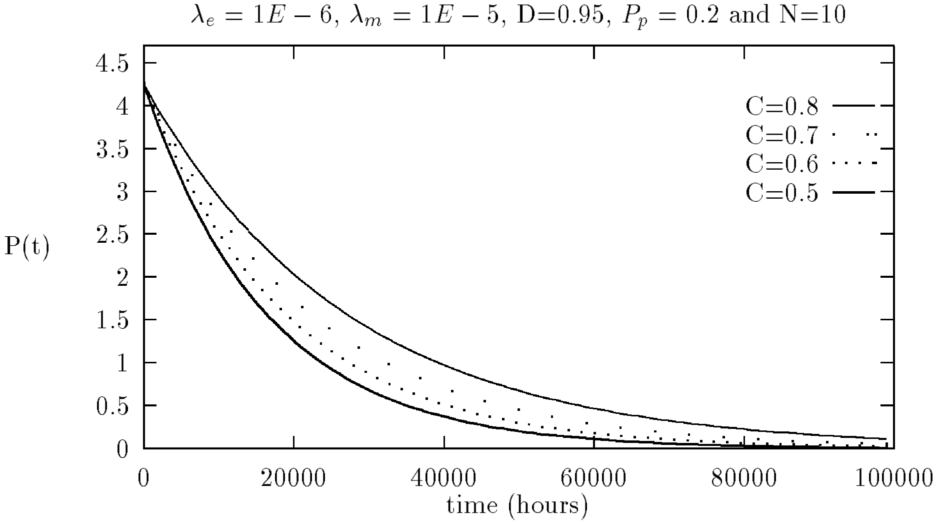


Figure 2.23: Influence of transient fault coverage on the performability of a non-degradable (CARER) multiprocessor system with checkpointing and recovery

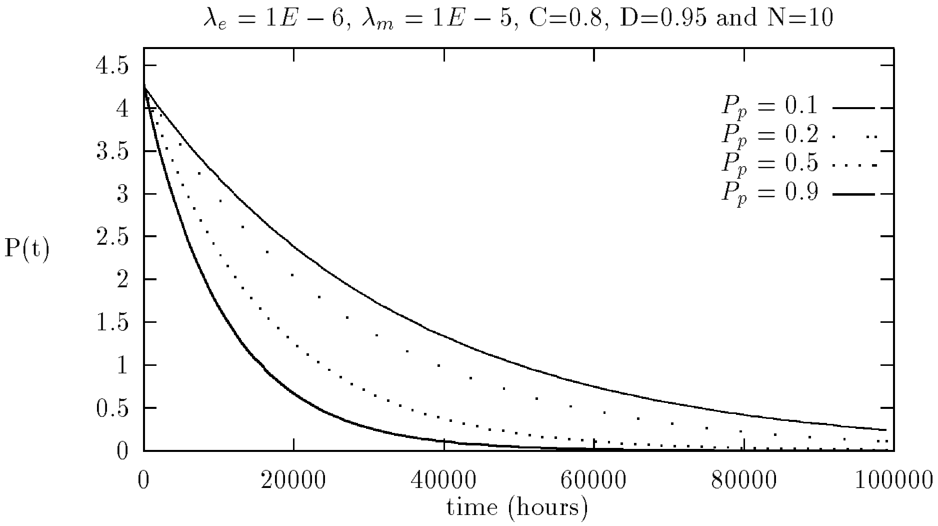


Figure 2.24: Influence of permanent fault rate on the performability of a non-degradable (CARER) multiprocessor system with checkpointing and recovery

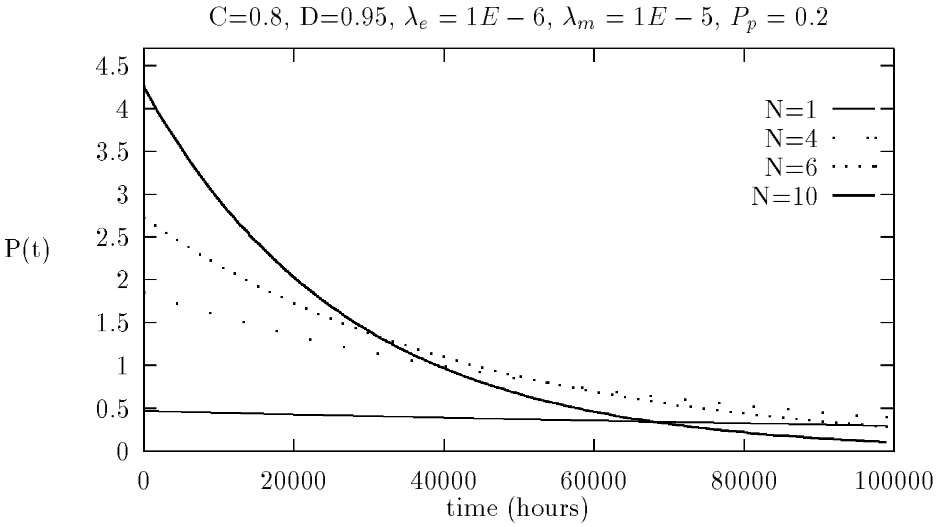


Figure 2.25: Influence of the number of processors on the performability of a non-degradable (CARER) multiprocessor system with checkpointing and recovery

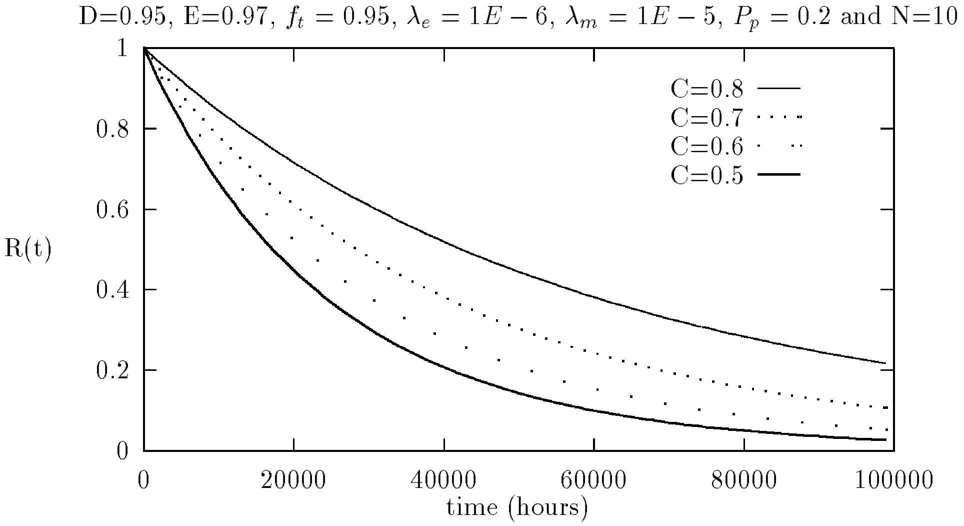


Figure 2.26: Reliability for a degradable (FASST) multiprocessor system with checkpointing and recovery

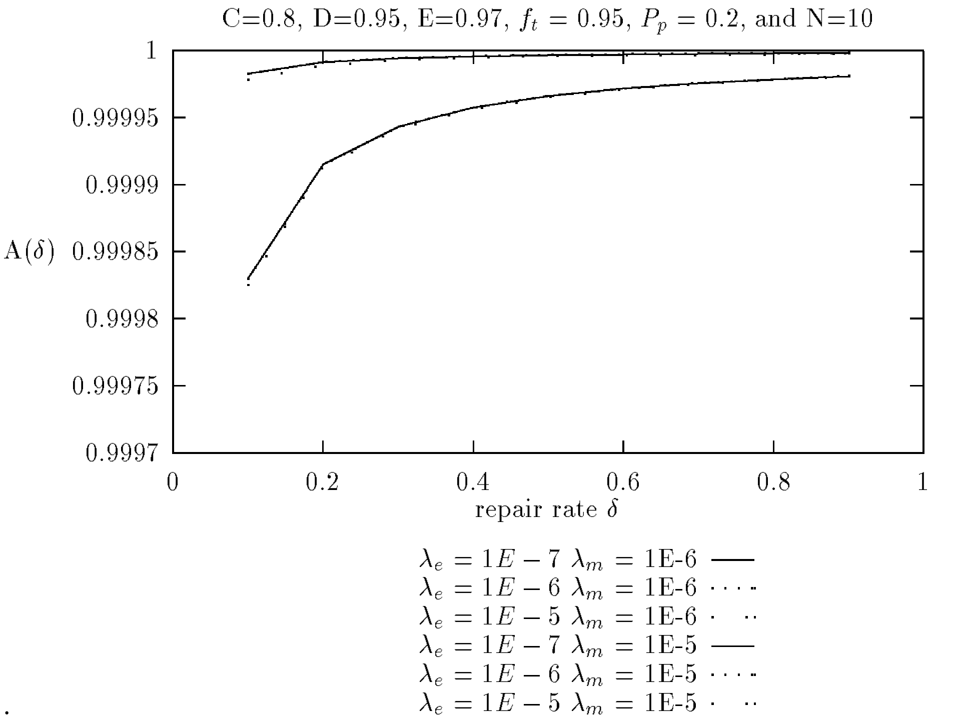
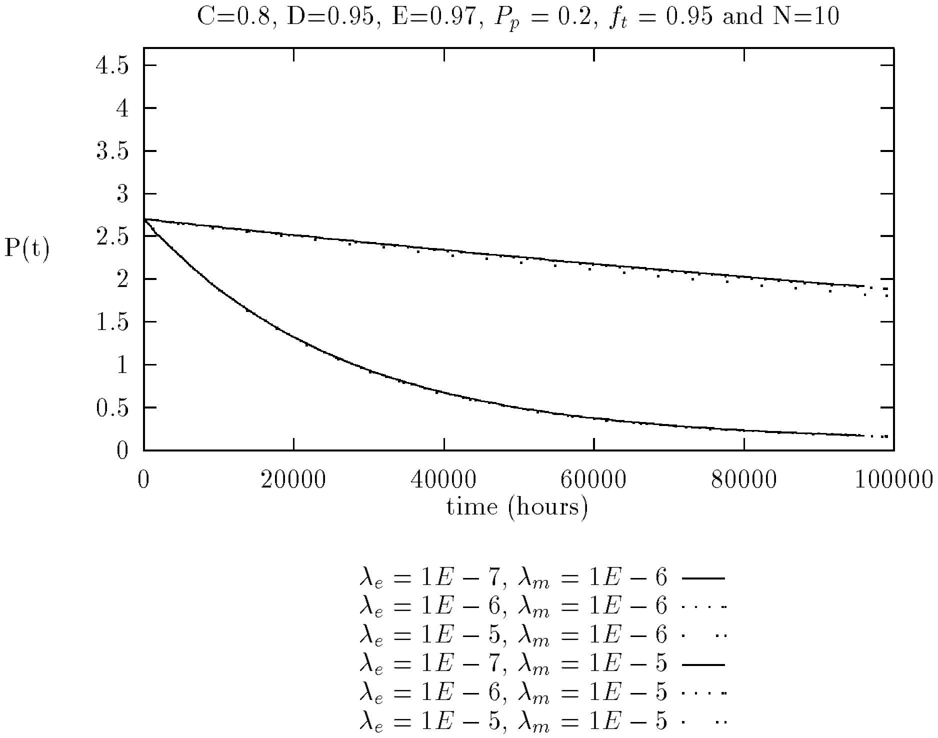


Figure 2.27: Availability for a degradable (FASST) multiprocessor system with checkpointing and recovery



.

Figure 2.28: Influence of processor and memory fault rates on the performability of a degradable (FASST) multiprocessor system with checkpointing and recovery

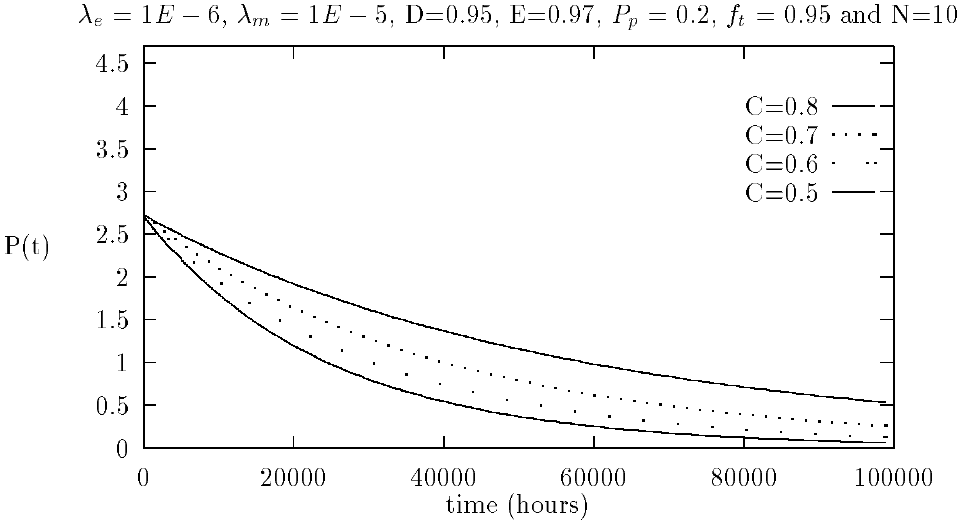


Figure 2.29: Influence of transient fault coverage on the performability of a degradable (FASST) multiprocessor system with checkpointing and recovery

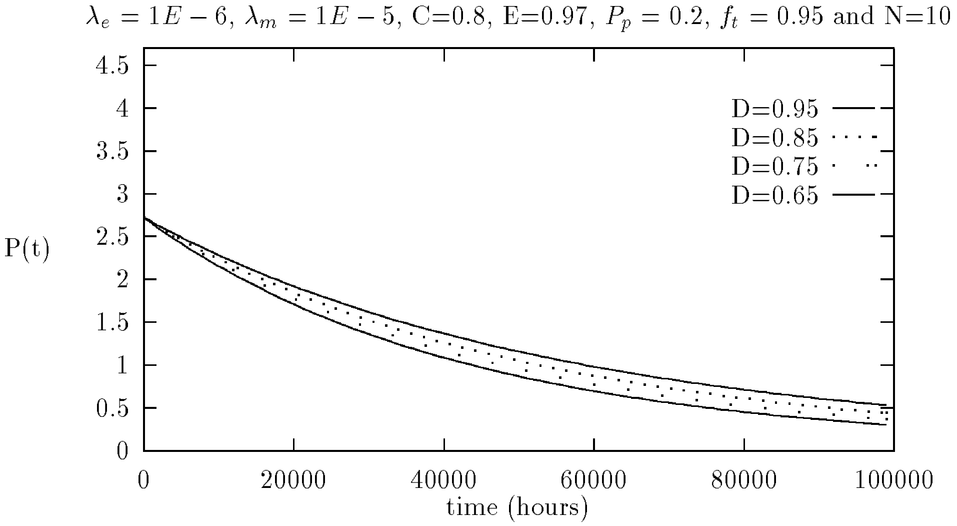


Figure 2.30: Influence of permanent fault coverage on the performability of a degradable (FASST) multiprocessor system with checkpointing and recovery

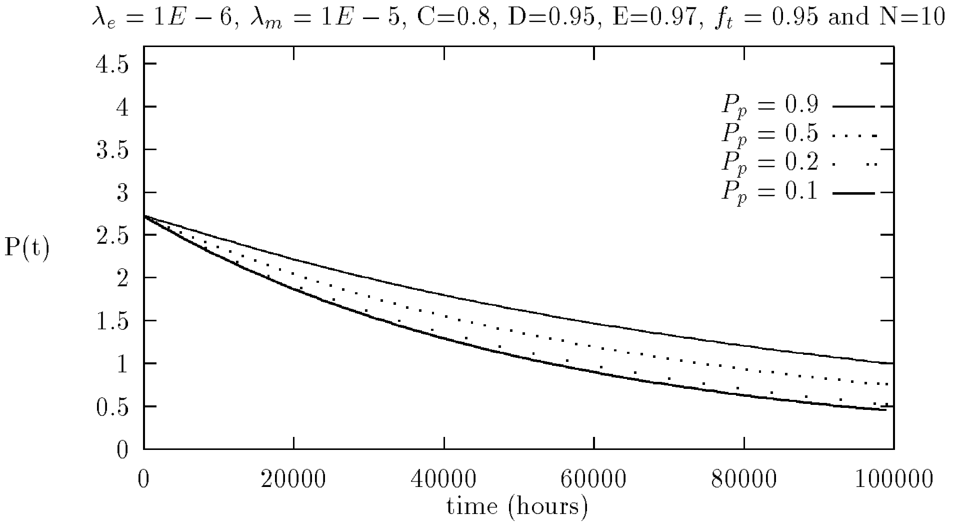


Figure 2.31: Influence of permanent fault rate on the performability of a degradable (FASST) multiprocessor system with checkpointing and recovery

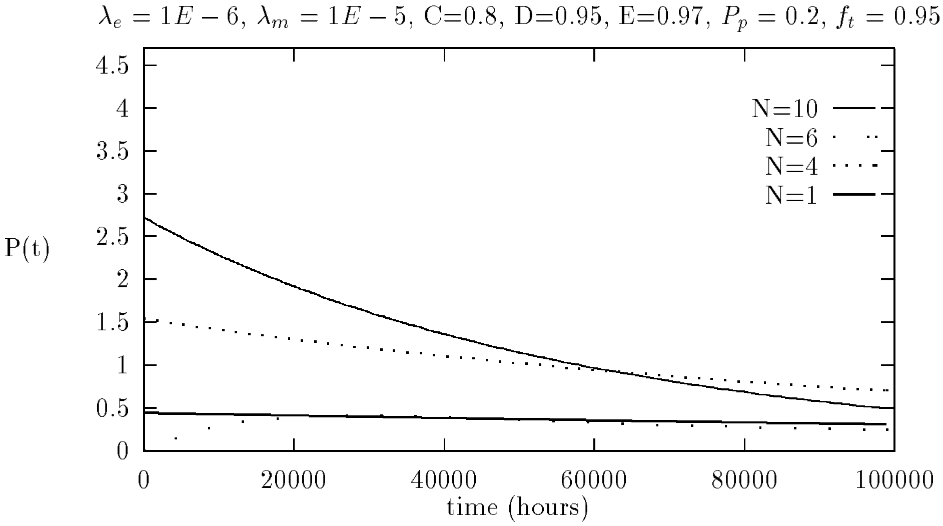


Figure 2.32: Influence of the number of processors on the performability of a degradable (FASST) multiprocessor system with checkpointing and recovery

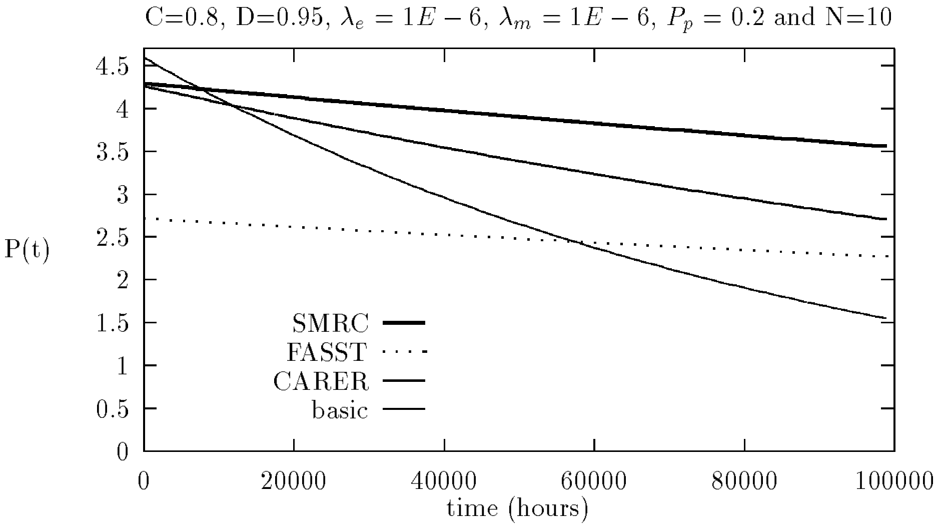


Figure 2.33: Performability comparison with memory fault rate $\lambda_m = 10^{-6}$

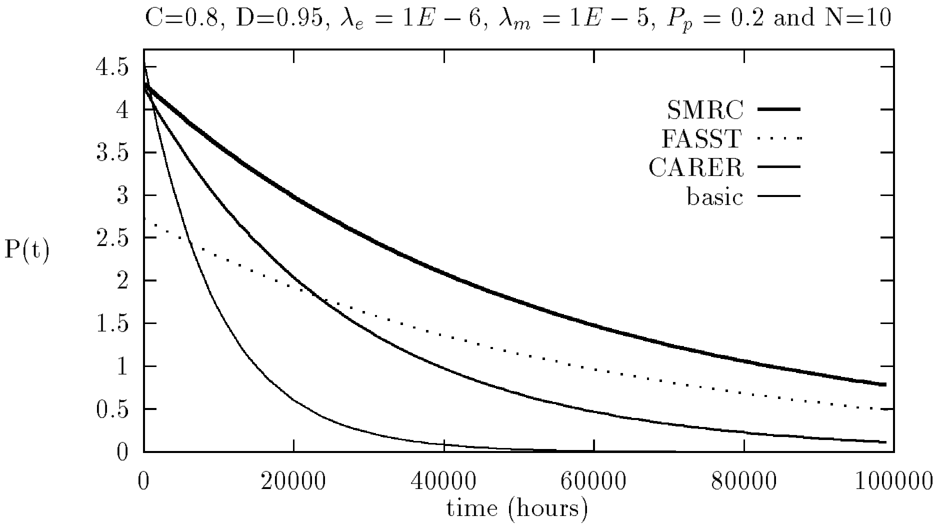


Figure 2.34: Performability comparison with memory fault rate $\lambda_m = 10^{-5}$

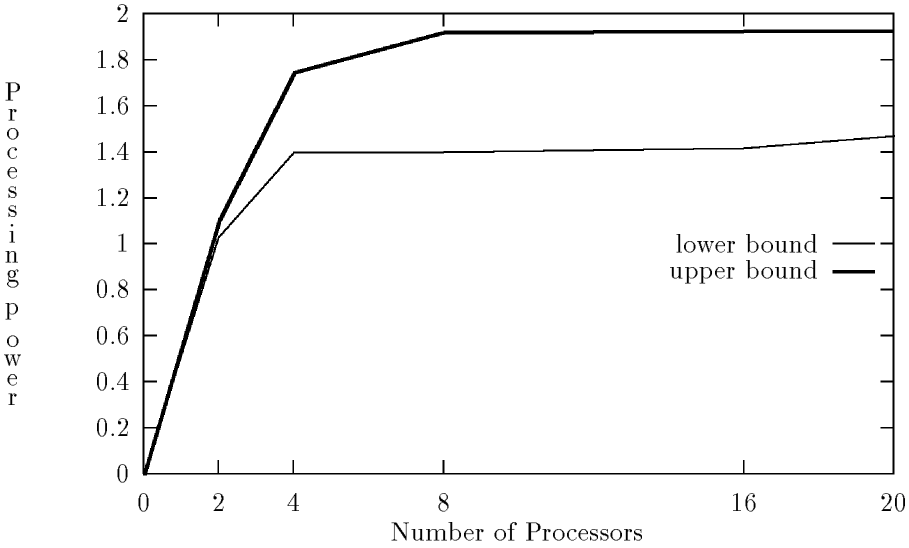


Figure 2.35: Processing power for cache coherent dual bus model using the *Write-Once* protocol

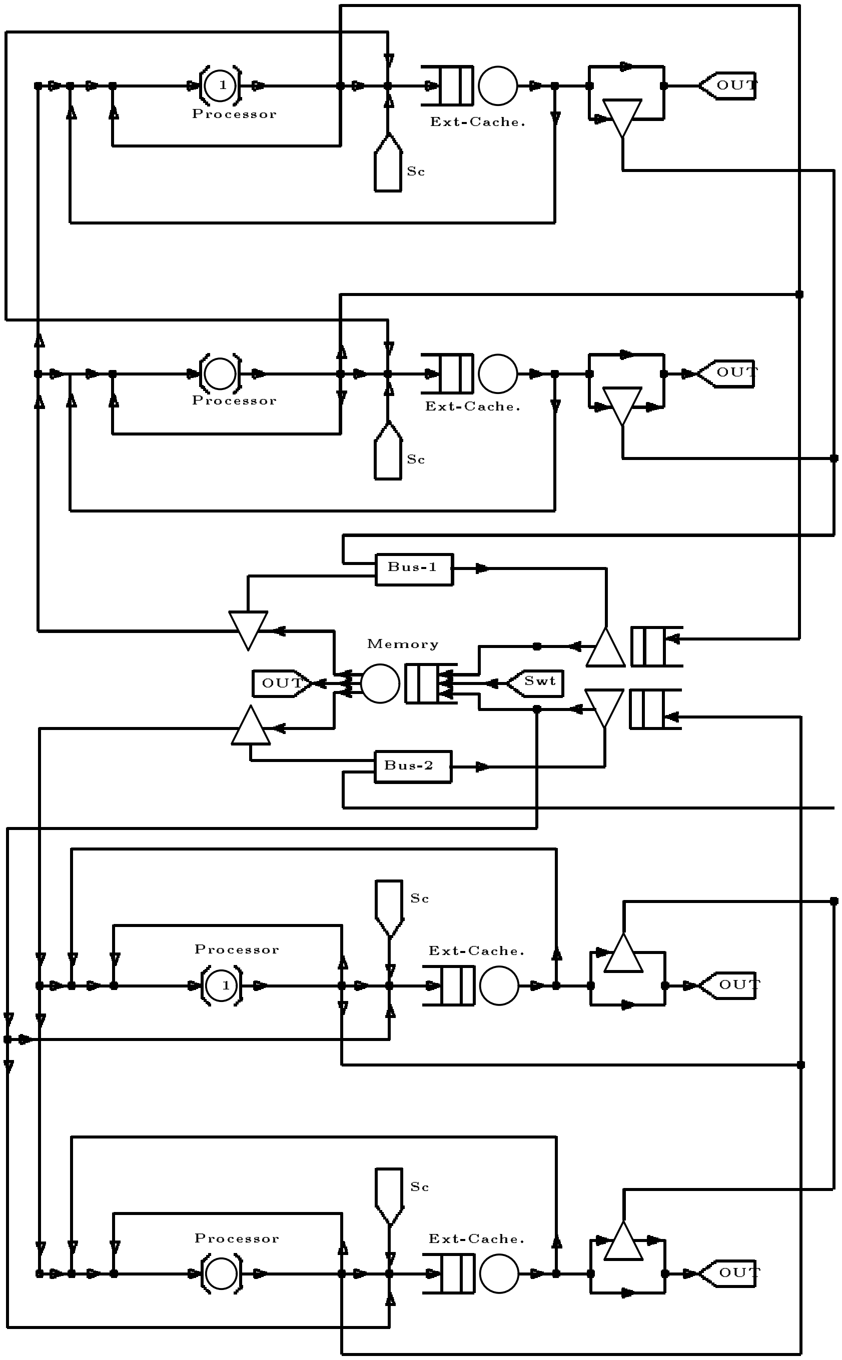


Figure 2.36: Upper bound model for cache coherent dual bus

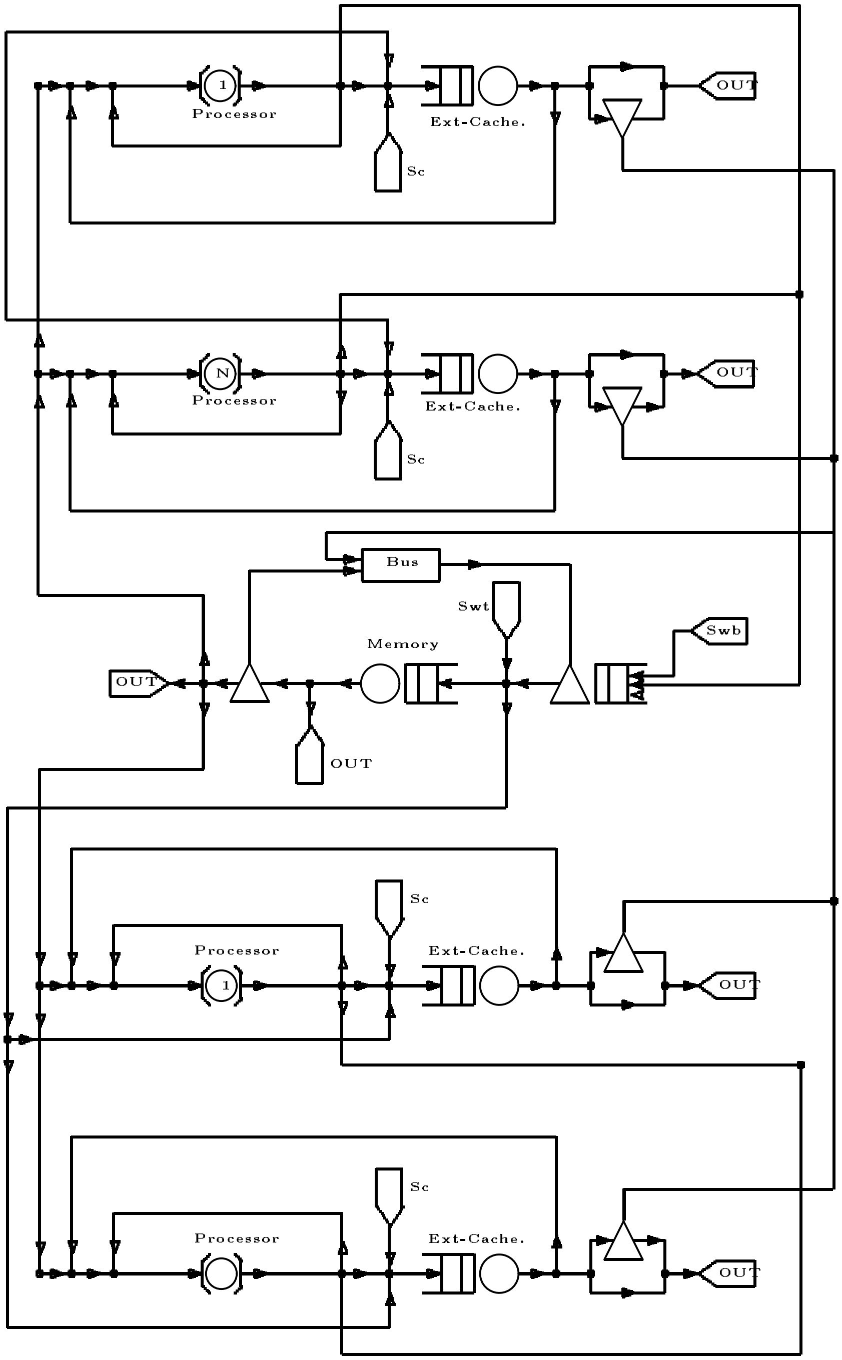


Figure 2.37: Lower bound model for cache coherent dual bus

Chapter 3

FASST Architecture

3.1 The FASST Architecture[1]

This chapter elaborates on the salient features of the FASST architecture, and then attempts to further model its performance. The general architecture of Figure 3.1, which is designed to transparently tolerate processor failures, mainly consists of processing elements, a bus, and a recoverable shared memory (*stable memory*, or SM) which provides normal memory functionality as well as a backward error recovery mechanism.

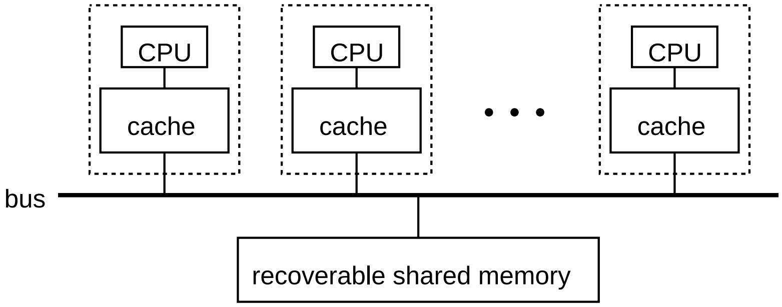


Figure 3.1: The FASST architecture

The architecture has been designed to require specialised hardware only for the SM. Standard processors, caches and cache coherence protocols can be used, and thus memory can be freely shared between processors. In particular, the recovery protocol avoids the use of dedicated blocking caches which require custom hardware and penalise the overall performance of the architecture. In principle, it should be possible to plug SM boards into an off-the-shelf shared memory multiprocessor to provide an error recovery mechanism for that system.

In the following, we further explain the FASST backward error recovery protocol that was introduced in Chapter 1 (see Section 1.4), and discuss how this protocol is implemented by the SM. The basic features and operation of the SM are introduced in 3.1.1 with the simplifying assumption that there are no caches interposed between the processors and the SM. The additional complexities added when snoopy caches are incorporated into the multiprocessor are discussed in 3.1.2.

3.1.1 Basic Features of the Stable Memory

As discussed in 1.4, the backward error recovery protocol has to permit recovery points to be established, recovered to and committed, and must permit a recovery line to be identified when recovery is required (also see [Banâtre et al 92a, Joubert 93]). The basic mechanism in the SM for providing recovery is to record recovery data for each memory location, essentially by maintaining two copies of each location. When a recovery point is established, each copy contains the same data. Subsequent updates to a location are made to only one of the copies and thus the second copy retains the state of that location at the time the recovery point is established. As only a single extant recovery point is needed for tolerating processor faults, only two copies of a location are ever needed.

To permit the identification of recovery lines it is necessary for the recovery protocol to:

(a) detect and record the existence of inter-processor dependencies which arise through their sharing of data in the memory; and

(b) synchronise the recovery protocol operations of those dependent processors.

The synchronisation is a vital part of the recovery protocol. In Chapter 1, for instance (remembering the final assumptions of Section 1.5), we showed that if processor P_j reads a cell previously modified by processor P_i within its current recovery region, then if P_i is recovered because the processor on which it was executing fails, then P_j must also be recovered. We also showed that if processor P_i writes into a cell previously modified by processor P_j within its current recovery region, then if P_i is recovered then P_j must also be recovered, since the value written by P_j has been overwritten by P_i and so cannot otherwise be recovered. In both previous cases, an

[1] This section contributed by : Michel Banâtre, Alain Gefflaut, Philippe Joubert and Christine Morin, IRISA/INRIA, Campus universitaire de Beaulieu, F-35042 Rennes cedex France, and Pete Lee, Department of Computing Science, University of Newcastle, Newcastle upon Tyne, NE1 7RU, UK.

extant recovery point is required for P_j so that the recovery line linking P_i and P_j actually exists. Thus dependent processors have to synchronise their actions on establishing, recovering to, and committing recovery points. One simplification can be obtained by ensuring that a processor always has an extant recovery point, by ensuring that a new recovery point is automatically established when a previous one is committed or restored. Thus the SM protocol does not provide a separate *establish* operation.

Clearly, a key part of the recovery protocol is in detecting and tracking interprocessor dependencies. This is achieved in the SM by means of recording *dependency relationships*. A dependency has to be recorded by the SM in two cases :

(a) Whenever a processor P_i reads a cell previously modified by processor P_j within its current recovery region.

(b) Whenever a processor P_i modifies a cell previously modified by processor P_j within its current recovery region.

A processor *P* is said to be the *active writer* of a cell *c* if *P* has written to *c* within its current recovery region and *c* has not been written to subsequently by another processor.

This dependency information is stored within the SM and used to compute recovery lines, or more specifically the *dependency group* of processors involved in the commitment or restoration of a recovery point. Recovery of a processor P_i must induce recovery of its dependency group; this requires the values of all of the memory locations which have been updated by any processor in that group to be restored by the SM to their prior values, using the recovery data recorded by the SM for this purpose. Similarly, commitment of a process will induce the commitment of all processors in the dependency group; in this case the values of all of the memory locations which have been updated by any processor in that group must be committed. This entails the SM in making the two copies of the memory location identical.

One important assumption for dependency tracking is that the SM is only connected to the bus of the architecture and dependencies are tracked by $snooping$ bus information, i.e. it requires *broadcasted* information; non-broadcast mechanisms (for instance some of those used in distributed shared memory systems) will require redefinition of the recovery protocol. In order to record dependencies, when a memory location is accessed by a processor, the SM needs the following information:

(a) The identity of the processor performing the access. This requires that each processor has a unique identifier which is transmitted whenever that processor generates a read or write request.

(b) The type of access (read or write)

(c) The identifier of the processor that is active writer of the cell if any.

The commitment of a recovery point by a processor obeys a simple distributed *two-phase commit* [Gray 78] protocol. Processors are participants while the SM is the coordinator of the protocol. In contrast to the standard two-phase commit protocol where the coordinator is responsible for triggering the protocol, it is a participant which initiates commitment; yet it is the coordinator itself which is responsible for actually committing data. When a participant wishes to issue a commit request, it must first flush its internal registers out to the SM (since the values in these registers, which form part of the global state, cannot otherwise be accessed). It can then send a *do_commit* command to the SM and wait for an interrupt signalling that commitment has terminated and that the processor can resume processing.

Upon receiving a *do_commit* command, the SM scans its dependency information (to determine the recovery line) and informs all of the processors in the dependency group. A dependent processor can then flush its registers into the SM if necessary and must acknowledge its completion of the first phase of the protocol. When all acknowledgements from the participant processors have been received, the SM enters the second phase of the commit protocol. During this second phase, the recovery data of the cells whose active writers belong to the dependency group is discarded. Once this has been achieved, commitment is complete and a new recovery point is established for each processor belonging to the group. Thus the processors in the group are no longer dependent upon each other, and the dependency information in the SM can be discarded and the participants allowed to proceed with their computations.

Let us consider now the implementation of the SM in greater detail.

Servicing read and write requests

The SM actions are best described by a finite state automaton. The automaton includes an $initialisation$ state together with a $service$ and $commit\_phase2$ states. In the service state of the SM, most of the work is concerned with dependency management. Assume that dependency data is stored in a $n * n$ boolean matrix M; n being the maximum number of processors in the architecture. A matrix item $M(i, j)$ is set to true when processor P_i is dependent on P_j.

While read and write requests may refer to SM cells, the SM itself may record dependencies on a larger granularity. In the following, it is assumed that the SM physical space is divided into a set of contiguous $blocks$ of identical size that is a power of two of the cell size. There is no constraint that the block be, say, the size of a cache line or a memory management page; it may be either or neither. Each block consists of:

- **(a)** current values of the cells,
- **(b)** a tag field containing either the identity of the active writer to the block (if any) or the nil value, and
- **(c)** recovery data.

Basically a read request involving cell c requires the SM to compute the identity of the containing target block b, record a dependency between the processor making the request and the active writer of the block (if any) in the matrix M, and deliver the current value of the cell. A write to a cell c will compute the target block b, record a dependency with the active writer if any, change the active writer of the block, and update the current value of the cell within the block b.

First phase of the commit protocol

Upon receiving a *do_commit* command from processor P_i, the SM has to scan the dependency information it has recorded in matrix M during P_i's current recovery region to determine the *group* of processors which are required to commit atomically with P_i according to the recovery protocol. Once the dependency group has been computed, each processor in that group has to be informed that it is required to participate in this commitment, by means of a *prepare_to_commit* interrupt. This can be implemented in a variety of ways, using interrupt or message passing facilities provided by the bus. The SM could generate such interrupts directly. Alternatively, the SM could broadcast on the bus a bit vector conveying the group of dependent processors, with dedicated logic on each processor board checking whether the processor it is attached to has to participate to the group and generating the *prepare_to_commit* interrupt appropriately (this checking could also be implemented in software). Each processor in the dependency group must also issue a *do_commit* command in acknowledgement, meaning that as far as it is concerned, the first phase of the commit protocol is OK.

It should be noted that within the interval between the initial *do_commit* command and the receipt of acknowledgements from the dependent processors, some new dependencies may have been created, since the SM can continue servicing read and write commands from processors that are not blocked waiting for the end of the commit protocol. These processors have to be added to the dependency group if this concurrency is permitted. It may also occur that a processor not already part of the group decides to commit its current recovery point and sends a *do_commit* command to the SM. This processor is added to the (current) group as well as all the processors dependent upon it. This mechanism provides a simple means for implementing multiple concurrent processor groups.

One crucial point is that computation of the dependency group has to be atomic with respect to read and write accesses to the SM. If it is not atomic, the group could be incorrectly calculated by the SM. A simple way to implement this atomicity is to serialise the group computation and normal read and write accesses.

The dependency group computation algorithm is given in the C programming language in Figure 3.2, with the assumption that the number of processors n can be encoded within an integer variable, and the matrix M is implemented by an integer array where each array element is considered as a bit vector indexed by a processor identifier. Upon reception of a *do_commit* command, the SM executes the *do_commit* procedure. The bit vector $group$ denotes the processors which are members of the dependency group, while $do\_commit\_received$ is the bit vector denoting the processors that have completed the first phase of the commit protocol. The *do_commit* procedure of Figure 3.2 will cause a state transition of the SM automaton into the $commit\_phase2$ state that implements the second phase of the commit protocol if the following condition is verified :

$$Q:\ ((group = do\_commit\_received) \wedge (\forall i:\ i \in group : immediate\_ancestors(i) \in group)$$

This condition expresses the requirement that all processors belonging to the group of dependent processors have completed the first phase.

Q may not be satisfied in two situations. Firstly, some processors that have already been informed that they are group members have not yet completed the first phase, in which case the SM must wait for the reception of their *do_commit* acknowledgement commands. Secondly, some new processors have become group members since the last computation of $group$ and thus must be informed. Notice also that the dependency computation algorithm must avoid interrupting a given processor more than once.

```
    int state;               /* current state of the automaton */
    int group;               /* dependency group computed so far (bit vector) */
    int do_commit_received;  /* bit vector of do_commit commands */
                             /* received from the processors */
    int M[n];                /* dependency matrix */

INITIALISATION:
    do_commit_received = 0; group = 0;
    for(j=0;j<n;j++) M[j] = (1<<j);   /* a processor is an ancestor of itself */
    state = SERVICE;

SERVICE:
    read(address) {
      /* create a dependency if necessary and return the value stored at address */
    }

    write(address, value) {
      /* create a dependency if necessary and perform the write */
    }

    do_commit(int i) /* i is a processor id */
    /* the  processor i is willing to commit or acknowledges a request of
       the SM following a commit request from a dependent processor */
    {
      int dependent_members;
      int j;                 /* processor id */

          /* add processor i to the group */
              group |= (1<<i);
          do_commit_received |= (1<<i);

          /* compute new dependent members */
              dependent_members = group;
              for(j=0; j<n ; j++)
                { if ((group & (1<<j)) != 0)     /* if processor j is a member */
                                                 /* of the group */
                     dependent_members |= M[j];  /* add immediate ancestors */
                }
          /* {dependent_members = group ==> group is exact} */

          /* check for termination condition Q and inform new members if necessary */
              if ((do_commit_received == group) && (dependent_members == group))
                  state = COMMIT_PHASE2;
              else if (dependent_members != group)
                      {  /* broadcast (dependent_members&~group) onto the bus */
                         group = dependent_members;
                      }
     } /* do_commit */
```

Figure 3.2: Computing a dependency group

There are many ways to devise an algorithm satisfying the previous requirements. In Figure 3.2, a simple solution is given. The algorithm checks the Q condition and as a side-effect computes a new value of $group$. If the new value of $group$ is different from the last value, the new members are informed. The complexity of the algorithm is $O(n)$, with the *do_commit* procedure being executed at most n times. The first phase of the commit protocol is thus $O(n^2)$. Note that termination is obvious from Figure 3.2 assuming that a processor acknowledges a *prepare_to_commit* request within a finite time and since the number of processors is bounded.

Second phase of the commit protocol

The basic actions which have to be performed in the second phase of the commit protocol are the following:

- **(a)** Discard the recovery data of all the blocks whose active writers belong to the dependency group and establish a new recovery point. This can be achieved by setting the active writer field of those blocks to the *nil* value and by setting the recovery data values to the current values.
- **(b)** Break the dependencies by updating the dependency matrix M appropriately.
- **(c)** Broadcast a *commit_done* interrupt to the processors belonging to the dependency group to permit them to restart computations.

As in the first phase of the commit protocol, these operations need to be atomic with respect to read and write accesses.

The implementation of the second phase of the commit protocol has a great impact on the overall performance of the architecture since a processor must not modify a block for which the new recovery data has yet to be recorded. Several implementations of the second phase can be devised. A trivial solution would consist of sequentially checking every block of the SM and copying the current value of the block to its recovery counterpart if necessary. This leads to the second phase of the commit protocol taking a time proportional to the size of the SM, which may not be desirable. Moreover, the processors must be prevented from restarting before this copying has been completed.

Several refinements to this straightforward algorithm can be made. The SM could maintain a per processor linked list of modified blocks. The time needed to perform the second phase is then proportional to the number of memory blocks which had been updated by the processor group, but at the cost of extra storage within the SM.

Alternatively, it is possible to permit the SM to restart normal memory operations, and to delay the effective copying of a block (to provide recovery data for the new recovery point) until it is really needed, that is, if a processor attempts to modify that block. This copy on write mechanism allows the second phase time to be interleaved with normal processor accesses, and processors do not need to be stalled until all the copying has been done in one fell swoop. However it is still necessary to mark blocks within the SM to determine whether a given block has to be copied or not on a subsequent write access. One approach is to use *checkpoint identifiers* [Wu et al 90]. In this approach, a checkpoint identifier is associated within the SM with each block and with each processor. When a block is modified, the current value of the checkpoint identifier of the active writer is stored along with the block. When a processor commits, its checkpoint identifier is incremented. Upon each write access, if the checkpoint identifier of the block is less than the checkpoint identifier of its last active writer, the current value of the block is needed for recovery data and so needs to be copied. Before allowing the write to perform, the block is copied to its recovery counterpart and the active writer and checkpoint identifier fields of the block are set accordingly. Similar optimisations were provided in various implementations of the recovery cache [Lee et al 90].

Tolerating processor failures

In this paper we have assumed that the processors used are *fail-stop* [Schneider 87], and that a failed processor can be easily identified. This is not a severe constraint on the architecture, for fail-stop processors are common practice in the field of hardware fault tolerance (e.g. through the use of duplicated processors). In case of failure, the processor ideally will signal a failure interrupt on the bus which will be caught by one of the live processors. This processor can trigger the recovery process by issuing a *do_rollback(i)* command to the SM, where i denotes the processor that failed.

Upon reception of the *do_rollback(i)* command, the dependency group of i is computed by the SM, thus identifying the group of processors which must be recovered in order to reset the system state to a consistent state, that is the state at a recovery line, in a manner similar to the second phase of the commit protocol discussed earlier. However, the values of the blocks modified by the members of the dependency group have to be reset to their prior state by copying the values held in the recovery data associated with those blocks. Each dependent processor must be interrupted by a *roll_back* interrupt to cause them to abandon their current processing. The dependencies are broken by resetting matrix M appropriately, and the SM reenters the service state. Recovering back to a recovery point is a simple protocol requiring a single phase compared to the commit protocol which requires two phases.

A particular situation may occur if group commitment is in progress when recovery is demanded. Since the same processor may belong both to a recovery group and a commit group, it is necessary to check for this at the end of the recovery procedure. Members of the recovery group are removed from the commit group. If the remaining commit group is not empty, the do_commit procedure of Figure 3.2 is executed taking one member of the commit group as an argument.

Finally, after recovery has taken place, the global system state is consistent, and the processors which have been recovered can recommence execution of normal computations. The computation that was running on the failed processor can be re-executed on one of the remaining processors and hence a system failure will have been averted. Moreover, the processor failure will have been tolerated transparently, since no alterations were required to the software of the application to provide these tolerance actions.

3.1.2 Influence of cache coherence protocols

Now we know the basic operation of the SM and how the recovery mechanisms work, we must examine what the complexities arise when the architecture contains coherent caches, as is the case with any realistic multiprocessor. As we shall see, the primary changes needed to the SM concern the dependency tracking mechanisms, so we must examine the influence of the *cache coherence protocol* on these mechanisms.

Most hardware cache coherence protocols proposed so far rely on the fact that broadcasted bus traffic can be monitored (*snooped*) by all caches attached to that bus; the remainder (non-broadcast protocols such as directory protocols, or some of those used in distributed shared memory systems) behave differently. Snoopy caches maintain a *tag field* stored along with each loaded line to indicate the state of that line. The tag field generally encodes whether the line has been modified with respect to the contents of the corresponding shared memory location, and whether the line has been loaded into another cache. As discussed in Sections 2.1 and 2.2, two main classes of snooping cache coherence protocols can be distinguished, depending upon the actions performed by caches when a shared line is modified. *Write Invalidate* protocols cause an invalidation message to be broadcast on the bus whenever any data potentially loaded into other caches is updated, to cause the cache lines to be invalidated elsewhere. *Write Update* protocols broadcast the new value whenever data potentially resident in other caches is updated. As in Section 2.1.1, we shall examine the *Berkeley* protocol [Katz et al 85], as a representative of the write invalidate family of protocols. In the *Berkeley* protocol, a cache line can be in one of the four states: Invalid (I), Non-modified Shared (S), Modified Exclusive (M), and Modified Shared (O), as shown in Figure 2.1.

Recall that the SM maintains dependencies on memory blocks. In contrast to the previous section, where the block granularity could be as small as a SM cell, when caches are present the SM must record dependencies on at least a cache-line size granularity, since a cache-line is the minimal unit of transfer on the bus. Therefore, let us examine the operations performed by the cache protocol and the various actions taken by the SM so as to track the dependencies when a processor performs respectively a read miss, a write hit, and a write miss on its cache (a read hit does not generate any action on the bus and thus does not need to be considered further).

Processor P_i performs a Read Miss If there exists a cache with a copy of the line in state M or O, this cache must supply a copy of the line to the requesting cache and set its state to O. Otherwise the line must come from shared memory. In both cases, the line is loaded in state S in the requesting cache. If the target block containing the line has an active writer P_j, a dependency must be created in the SM between P_j and P_i. As far as dependency management is concerned, no distinction is made whether the requested line comes from another cache or from the SM, although the inter-cache transfer must be detected by the SM snooping on the bus.

Processor P_i performs a Write Hit If the line is already in state M, the write proceeds without delay. Otherwise, (in state S or O) an invalidation signal must be sent on the bus. All other caches invalidate their copy if they have one that matches the line address. The line state is changed to M in the originating cache. The invalidation signal is snooped by the SM. If the corresponding block has no active writer, P_i becomes its active writer. Otherwise, if P_j was the active writer, a dependency is created between P_j and P_i and P_i becomes the active writer of the block.

Processor P_i performs a Write Miss Like a read miss, the line comes from its owner or from shared memory. All other caches invalidate their copy if any. The line is loaded in state M. The SM snoops the data transfer if the line comes from another cache. As above, if the corresponding block has no active writer, P_i becomes the active writer; otherwise, a dependency is created between P_i and P_j and P_i becomes the active writer of the block. Since cache lines can contain several processor addressable cells and the line is now cached by P_i in state M, the SM cannot detect a further read on a different cell of the line because it would not generate any bus traffic. So, a dependency between P_j and P_i is also created to prevent the case in which a cell previously modified by P_j would be locally read by P_i. In other words, the SM adopts a conservative approach by creating some dependencies which are not strictly required by the protocol to preserve the coherence of processor checkpoints.

It should be noted that the SM must keep pace with the information exchange rate on the bus due to the cache coherence protocol. If this were not the case, the SM might miss some dependencies that need to be recorded.

The commitment of a recovery point when caches are present is similar to the situation where no caches are present. What is required is that when a participant processor initiates commitment or acknowledges a *prepare_to_commit* request from the SM, the processor must flush its cache as well as its internal registers. Similarly, recovery must cause a cache invalidation.

In summary, no special purpose caches or coherence protocols are needed in the architecture being presented here, which can accommodate standard cache behaviour with the SM performing dependency tracking by snooping the bus traffic. This is a notable difference with other proposals for fault tolerant shared memory multiprocessors [Bernstein 88, Wu et al 90, Ahmed et al 90].

3.2 Performance Evaluation

In the light of this, we may now extend the performance evaluation to a shared memory multiprocessor machine that incorporates a SM, i.e. to the FASST architecture. Through simulation, the performance of FASST has been compared against the performance of a standard multiprocessor architecture without any fault tolerance capabilities and against that of two other approaches for fault tolerant shared memory multiprocessors, namely CARER and Sequoia.

3.2.1 Methodology and workload

The simulations were conducted using an instruction level simulator driven by a set of memory references generated by instrumenting application code with the Abstract Execution technique [Larus 90]. The simulator implements an efficient execution driven simulation method similar to that described in [Davis et al 91] (further information on the simulation tool may be found in [Gefflaut 92]). Execution driven simulation controls the address trace generation to ensure that the trace corresponds to that which would be obtained if that application was actually executed on the architecture being simulated. This technique thus supports the derivation of simulations which accurately model the architecture. The simulation models were parameterized with the characteristics of Sun multiprocessor SparcServers, with a 320MBytes/sec synchronous bus, 64KBytes unified direct-mapped caches with 32 bytes lines and IEEE write invalidate cache coherence protocol. To simplify the performance comparison, all of the fault tolerant architectures were modelled with these parameters in common.

For FASST, the error recovery protocol is that described in Section 3.1. For the second phase of the commit protocol the stable memory implements the copy on write mechanism described in Section 3.1.1. Since the extent of recovery regions in the FASST architecture is not controlled by any hardware or application parameter, it is necessary to fix a rate for the frequency of recovery point establishment (and hence commitment) for the simulations. The only situation where FASST may be forced to commit a recovery point and to establish a new one is to prevent the loss or duplication of an operation on an unrecoverable object, for instance, I/O devices [Lee et al 90]. This classical technique for dealing with unrecoverable operations is used by CARER and Sequoia, and ensures that an I/O operation cannot be repeated. Thus, for the FASST simulations, each I/O operation leads to the establishment of a recovery point. To obtain an average I/O rate, the interrupt rate on a NFS file server was measured, and from this measurement a rate of 1000 interrupts per second was used in the simulations.

For the CARER simulation, a recovery point is committed and a new one established whenever a modified line in a cache needs to be replaced and whenever a modified line is read by a different processor. For Sequoia, recovery points are established and committed as described in Section 1.3, that is, whenever a blocking cache is full or a modified cache line needs to be flushed, or on exit from any critical section which requires coherence of the shared memory.

The workload comprises four parallel applications drawn from the SPLASH benchmark suite [Singh et al 91]. The application $cholesky$ performs sparse matrix factorisation; $mp3d$ simulates rarefied hypersonic flows; $pthor$ simulates digital circuits at the logic level; and $water$ simulates the evolution of a system of water molecules. Only the parallel phase of the computation was simulated, resulting in 65 to 80 million memory references for each application. The four applications were simulated on the four architectures for 1 to 8 processors.

3.2.2 Experimental results

3.2.2.1 Performance of the architectures

Figure 3.3 shows the MIPS (million instructions per second) performance of the four architectures for the four simulated applications. The performance degradation for FASST compared against the standard (non fault-tolerant) architecture is relatively small, despite a high commit rate for FASST (1000 per second). Performance degradation with eight processors remains below 15% except for $mp3d$ where it is about 30% (for reasons discussed below).

For the other fault tolerant approaches, the performance of CARER is relatively close to that of FASST for $cholesky$ and $water$ (10% performance degradation) but the degradation grows to 65% for $pthor$ and $mp3d$. CARER achieves these results despite the restrictive failure hypothesis (i.e. the caches are fault-free) that permit a very efficient implementation of its commit protocol. The Sequoia approach appears to offer the lowest performance of all three fault tolerant architectures. Performance remains below 100 MIPS independent of the application or number of processors used. The performance degradation for this architecture always exceeds 20% for one processor and can be as high as 85% ($pthor$ with 8 processors).

These results are very encouraging for the FASST approach to fault tolerance. Some degradation in performance over a non fault tolerant architecture is inevitable, due to the error recovery provisions in FASST architecture. Nevertheless, these simulations suggest a relatively modest degradation in general. When compared to the other fault tolerance approaches, the simulations suggest that the FASST approach provides the best overall performance.

3.2.2.2 Behaviour of the applications

It may be observed from Figure 3.3 that for all the architectures considered, the performance degradation varies significantly and is application dependent. Figures 3.4 and 3.5 show, for each application, the distribution of bus transactions for 10000 memory references:

- **(a)** misses serviced by shared memory;
- **(b)** misses serviced by caches;
- **(c)** write invalidations;
- **(d)** write backs arising from the replacement of a modified cache line (only for FASST since Sequoia and CARER use blocking caches); and
- **(e)** write backs arising from cache flushes for FASST and Sequoia or from the replacement of *unwritable* cache lines for CARER.

The figures also show the number of recovery points established for 10000 memory references. Three reasons for establishment are distinguished: the recovery points established because of interrupts (only for FASST), the recovery points established before replacing a modified cache line (for CARER and Sequoia) and the recovery points established because of data sharing. For FASST the latter are the recovery points established because of dependencies; for CARER they are established because of a miss on a cache line that has been modified in another cache; and for Sequoia they are the recovery points established upon exit from a critical section.

Cholesky

The standard architecture attains good performance for $cholesky$, with a speedup of 6.8 with 8 processors. This good behaviour is caused by a high cache hit rate of 99.2%. Data sharing is at a coarse granularity in this application. Although the 6% write ratio is comparable to other applications, the caches contain a low proportion of modified data due to the good locality of write references. Only 30% of replacements require a write back.

These characteristics allow performance degradation for FASST to remain always below 10% for this application. The caches do not contain a lot of modified data; on average with 4 processors, 360 cache lines are flushed at each commit. Also, the data sharing pattern of the application only creates a small number of dependencies (the average size of the group of dependent processors is 2.8 with 8 processors). These two factors explain the good performance of FASST for this application.

The performance of CARER is close to that of FASST for this application despite disproportionate recovery point establishment rates (CARER establishes 15 times more recovery points than FASST), due to the low cost of establishing a recovery point in CARER. Most recovery points are established when modified cache lines are replaced; only a few result from data sharing.

Sequoia suffers from an even higher recovery point establishment rate than CARER, mostly caused by critical sections that require frequent cache flushes. Moreover, the invalidation of unmodified cache lines on entry of a critical section contributes to lowering the hit rate from 99.2% to 98%.

mp3d

The behaviour of $mp3d$ is clearly worse than $cholesky$ for the standard architecture with a speedup of 5 for 8 processors. The cache hit rate is lower (98.3%) due to a worse write locality and to a 10% write ratio with 70% of replacements leading to write backs. Data sharing is very prevalent in this application; 77% of reads and 87% of writes reference shared data. Due to this heavy data sharing, half of the misses are serviced by caches. The large number of cache-to-cache transfers lowers the performance since a cache servicing a miss cannot service requests coming from its processor.

Of all four applications, FASST has the worse performance degradation for mpd3. Performance degradation is 13% for one processor and reaches 33% for eight processors. The major factor contributing to this result is the large amount of modified data resident in caches when a recovery point is committed; with four processors, an average of 1250 cache lines are flushed to memory. Moreover, due to the heavy data sharing, all processors are dependent. This considerably lengthens the duration of the first phase of the commit protocol.

CARER also does not behave well for this application because the heavy data sharing forces the establishment of a large number of recovery points. Moreover, a high number of modified cache lines are replaced.

Although synchronisation operations are infrequent, Sequoia suffers from the large number of modified cache lines replaced.

pthor

The standard architecture obtains low performance for the $pthor$ application, with a speedup of 4 for 8 processors. The cache hit rate is low (97.5%) due to a large working set, and the caches perform a significant number of write backs. Data sharing, although less intensive than for $mp3d$, contributes to limiting performance.

Performance degradation for FASST is much less for $pthor$ than for $mp3d$ (13% degradation for 8 processors). This behaviour is caused by the different amount of data flushed to memory when a recovery point is committed. With 4 processors, 570 cache lines are flushed, compared to 1250 for $mp3d$.

For CARER, although data sharing is less intense for $pthor$ than for $mp3d$ (38 cache-to-cache transfers vs 89 for 10000 memory references), the number of recovery points established because of data sharing is higher for $pthor$ than for $mp3d$ (12 vs 5 for 10000 memory references). This is caused by the data sharing pattern which is different for the two applications. In $pthor$, shared variables are accessed within short, but frequent, critical sections, thus leading to the establishment of a lot of recovery points.

The large number of locking operations and the large data set that causes a lot of replacements severely limits Sequoia performance for this application. No improvements of performance are achieved above 3 processors. Due to data cache invalidations upon exit of critical sections, the cache hit rate is lowered (84% instead of 97.5%).

water

The $water$ application offers high performance for the standard architecture because of a high hit rate (99.88%) and a small data set. Moreover, data sharing is negligible.

As might be expected, FASST performance is very good for this application because of the small amount of modified data resident in caches when recovery points are committed and of the few dependencies (2.6 dependent processors on average for 8 processors). CARER also behaves well for this application because of the small working set that only causes a few replacements of modified cache lines.

For Sequoia, only a small number of recovery points are established because of the replacement of modified cache lines, since the working set is small. Most of the recovery points are established because of critical sections.

3.2.3 Stable memory implementation

As there are several ways in which a SM could be implemented, it is important to consider the influence such implementations would have on the performance of a system incorporating a SM. Of particular concern is the potentially expensive operation of copying the current values of SM cells for use as recovery data whenever a recovery point is established.

Figure 3.6 shows the influence of this aspect of the SM implementation on performance degradation. Three implementations are considered: one using copy on write, one using a per processor list of modified memory blocks and the last which is a control case where the copying time is assumed to be nil, i.e. instantaneous.

As can be seen, and as might be expected, the different implementations greatly influence the performance degradation suffered by an application. However, the degradation ratio between the different implementations remains constant independent of the application. The copy on write implementation behaves better than the implementation using a per processor list of modified memory blocks, although the number of blocks to be copied (and so the time needed to copy those blocks) is the same for both. With the list of modified blocks, the duration of the copy is concentrated at the end of the first phase of the commit protocol. Although the processors can restart their computation at the end of the first phase, they are not allowed to perform bus transactions until all blocks have been copied, and so quickly become stalled waiting for the copy to be completed. If copy on write is used, the copying can be interleaved with normal memory accesses. Thus the processors are only kept waiting for short periods of time resulting in better overall performance. The performance is naturally the best for the control (instantaneous copy) case, which indicates the upper performance bound for the SM.

3.2.4 Dependency management

Dependency management adds some complexity to the implementation of the SM. In a simple implementation, all processors could establish a global recovery point, thus avoiding the burden of dependency management within the SM. In this case, a standard memory interface is sufficient for the SM since it no longer needs to snoop bus transactions to log dependencies. The commit protocol is also simpler. However, the potential gain of dependency tracking is in minimising the number of processors that have to be recovered in the event of a failure of one processor. Thus, it is useful to examine the impact of dependency management on performance to investigate whether it is worth the added implementation complexity.

Figure 3.7 shows, for each application, the performance degradation observed with eight processors, with and without dependency management. The figure also shows the average number of dependent processors at each commitment of a recovery point. For $pthor$ and $mp3d$, the performance of the two versions of the SM are nearly identical since for these applications all processors are dependent, and hence the presence of dependency tracking is irrelevant. In contrast, for $cholesky$ and $water$, where the average group size never exceeds three processors, the dependency management shows its efficiency since it reduces the performance degradation by a factor of two. The main reason for this is that with dependency management less data is flushed when a recovery point is committed, since fewer processors are dependent. For example, with the $water$ application, 190 cache lines on average are flushed to memory each time a recovery point is committed. When dependency management is suppressed, the number of cache lines flushed increases to 330.

3.3 Summary

These simulations have demonstrated that the CARER and Sequoia approaches to implementing a fault tolerant shared memory multiprocessor both exhibit similar performance behaviour. Both require the commitment of a previous recovery point and the establishment of a new recovery point each time a modified cache line has to be replaced, as well as when data sharing occurs (for Sequoia, data sharing is enforced explicitly by means of the locking protocol). The difference in performance of these two architectures primarily results from the differing costs of recovery point operations. A realistic implementation of CARER should consider the possibility of errors within caches and so would obtain roughly the same performance as Sequoia because of the consequent cache flushes. The rate at which recovery points are established and committed is controlled by cache parameters (size, associativity, replacement policy) and by the data sharing pattern of the application programs. This results in a high, uncontrollable and unpredictable frequency of recovery point establishment (between 25 and 100 times more than for the FASST approach). Some memory access patterns in the applications can even force the establishment of a recovery point at each data reference.

The FASST approach to implementing fault tolerance in a shared memory multiprocessor eliminates most of these disadvantages. The need for commitment/establishment of a recovery point is controlled primarily by the interactions of the architecture with its external environment (e.g. for I/O) independently of any architectural parameter. These interactions are much less frequent than cache line replacement. Recovery points are also independent of the communication patterns of the application programs, owing to the dependency tracking mechanism.

The fault tolerance overhead is concentrated in the commitment phase of the recovery protocol. Three factors can influence this overhead:

- **(a)** the amount of modified data,
- **(b)** the number of dependent processors,
- **(c)** the bus load of the machine.

The amount of modified data is the major factor that influences performance degradation. The duration of the commit is directly proportional to the amount of data that has been modified. In turn, the amount of data is governed by the number of processors that are dependent upon the processor that issued the commit. Thus, the dependency management mechanism in the SM minimises the number of processors that are affected by the commit (except of course if they are all dependent). The dependent processors impose another overhead on commitment, since some modified data may be resident in their caches, and commitment requires the modified lines in the caches to be flushed back to the SM. Thus, the importance of the dependency tracking mechanism in the SM increases with the number of processors in a system, in that minimising the number of dependent processors will minimise the amount of cache flushing and data committed. Note, of course, that cache flushing is required in an ordinary shared memory multiprocessor.

The bus load also influences the performance degradation. Performance degradation grows with the number of processors as does the bus load. Ideally the bus is lightly loaded, so that cache flushes can proceed without interfering with the activity of the processors that do not participate in the commit protocol, and in this case the performance degradation remains constant whatever the number of processors.

As stated in [Janssens et al 91], the key issue in obtaining good performance is to keep the frequency of recovery point operations independent of any architectural parameter. This is what the FASST approach attempts to do. The FASST architecture presented here allows processor failures to be tolerated transparently, that is, without affecting the software being executed on the architecture. The only specific hardware component required is the SM, and it is believed that the SM can be implemented at a reasonable cost. The SM copes with standard caches and cache coherency protocols, and this provides an advantage over the other approaches studied, such as CARER and Sequoia. The dependency tracking mechanism provided by the SM allows shared memory to be provided for and to be used by the software. In contrast, the Sequoia system only permits memory sharing within the operating system, and requires complex software structures to ensure the correct semantics.

From a performance point of view, simulation results show that the FASST architecture offers better performance than the Sequoia and CARER approaches, mainly due to a lower frequency of recovery point committments. Moreover, the amount of data copied in a commit operation is kept as low as possible by the fine-grained recovery protocol presented in Section 3.1 together with the dependency tracking mechanism.

The following chapters discuss the design of a SM, as well as a fail-stop *dual-processing unit* (DPU) and a *stable disk* (SD), before considering the system software issues. This chapter has only considered the case of parallel applications which consist purely of computation, with no input/output operations. Providing backward error recovery in the face of unrecoverable operations such as I/O is a further challenge. Extensions to the recovery protocol are necessary to provide the required abstraction of backward error recovery with both shared memory and other unrecoverable operations being executed by applications programs.

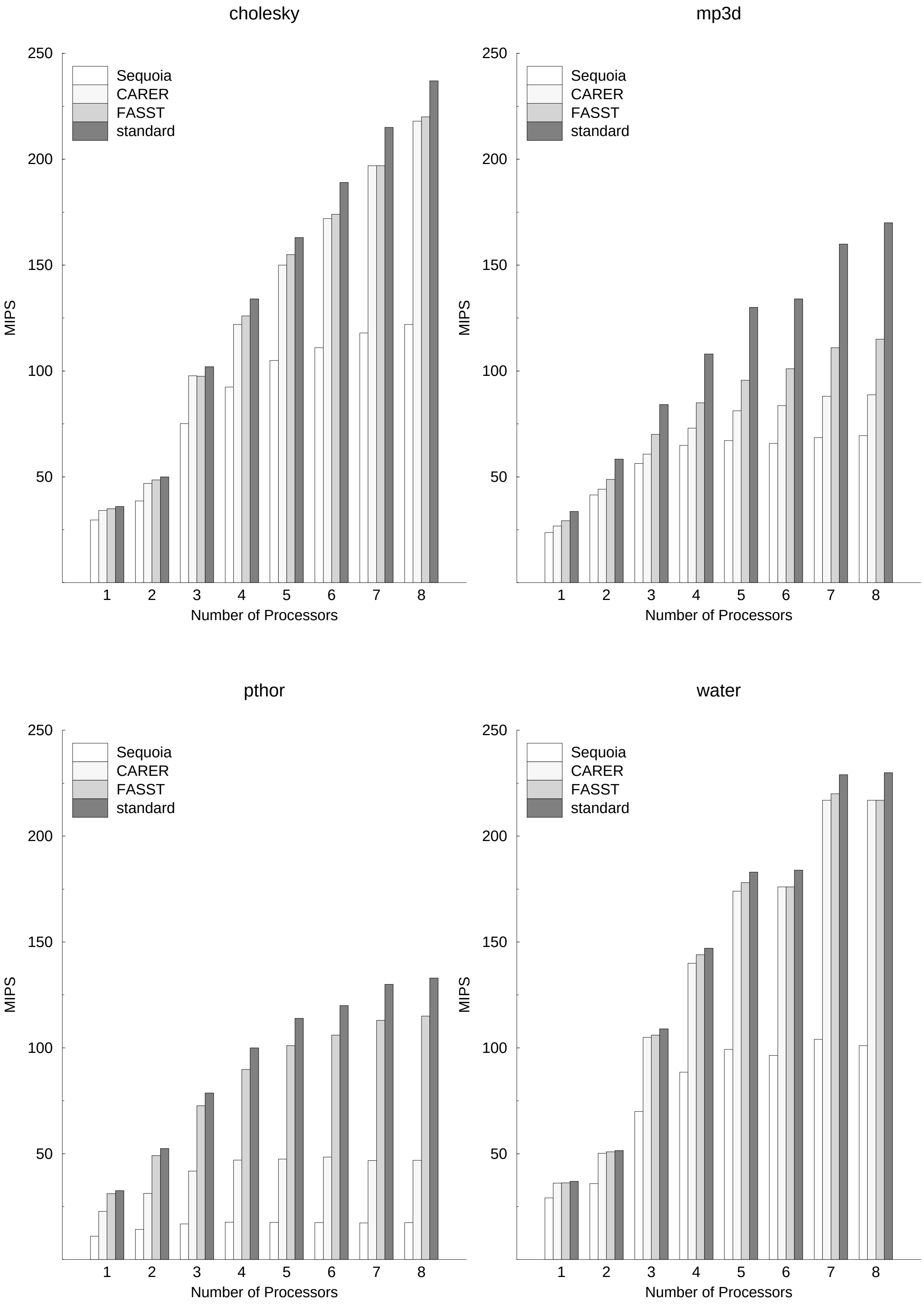


Figure 3.3: Performance

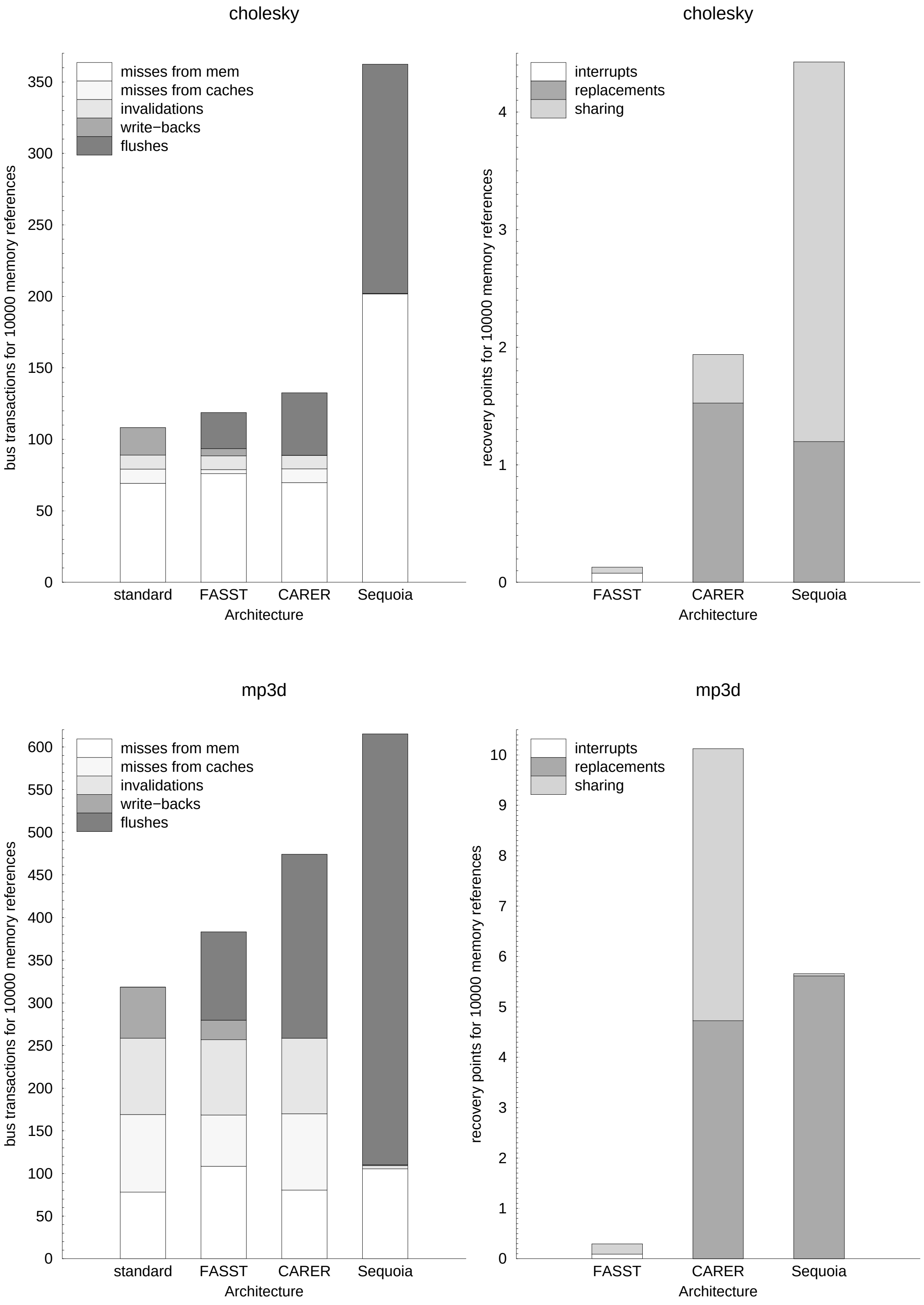


Figure 3.4: Application behaviour with 8 processors

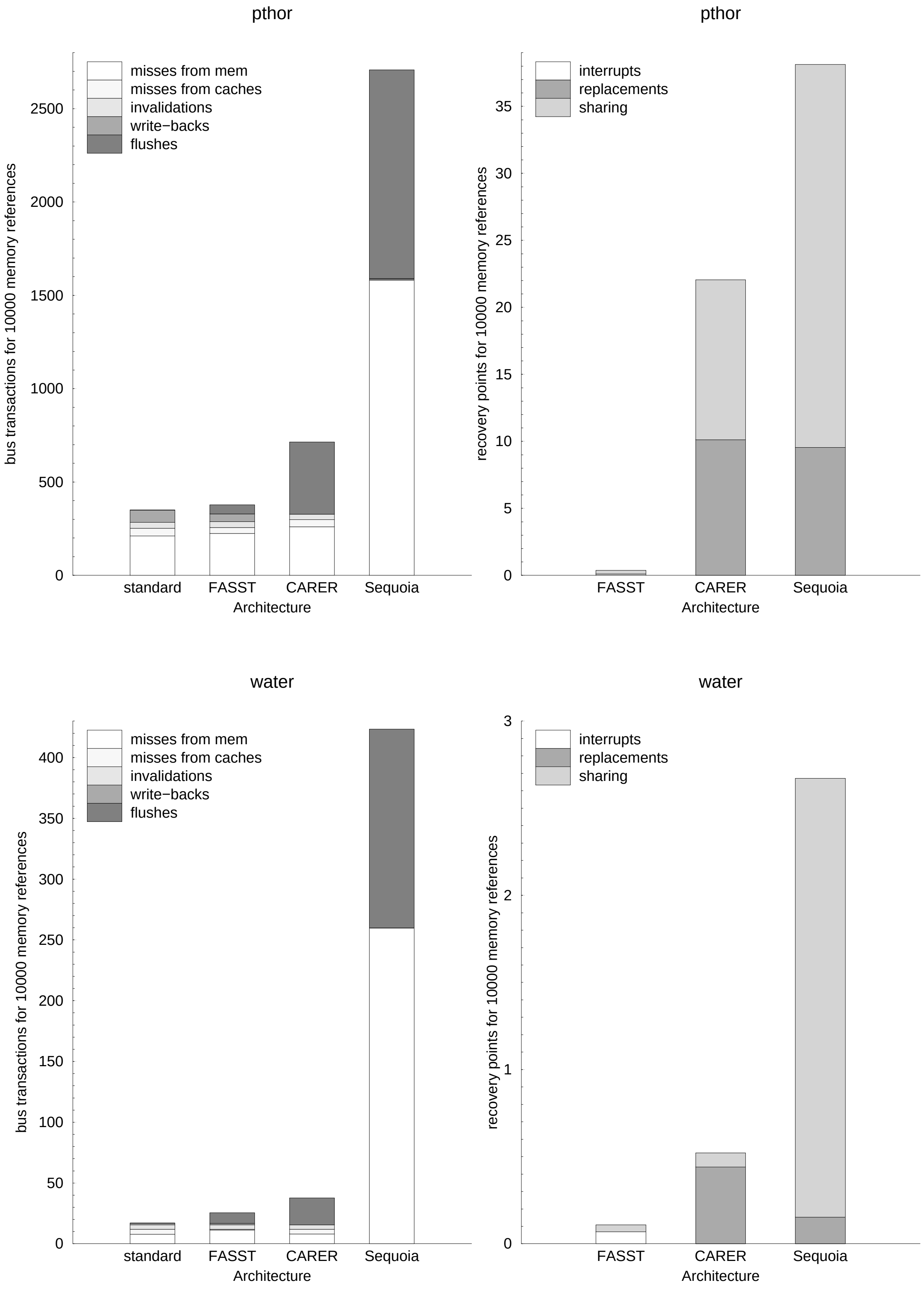


Figure 3.5: Application behaviour with 8 processors

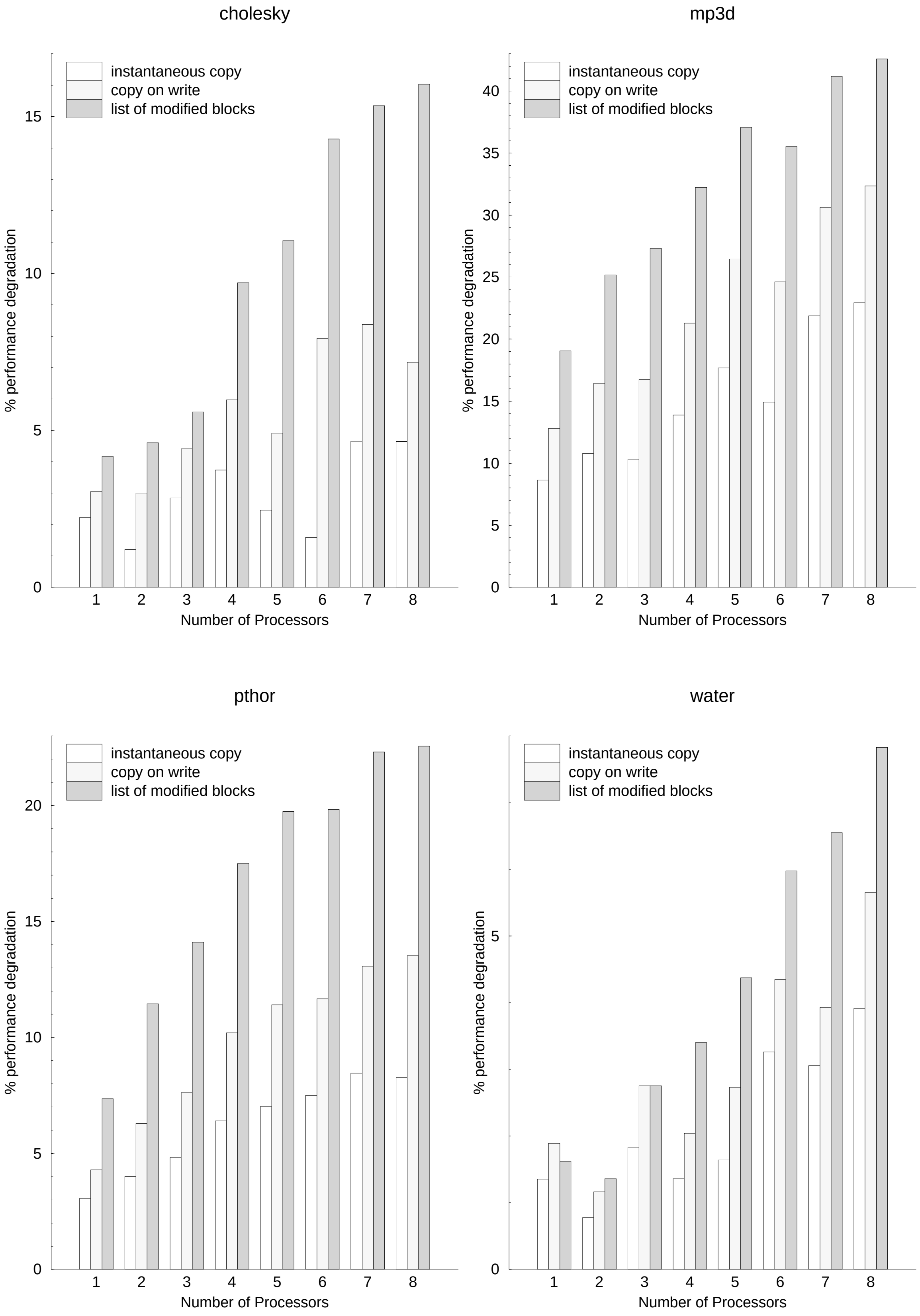


Figure 3.6: Stable memory implementation

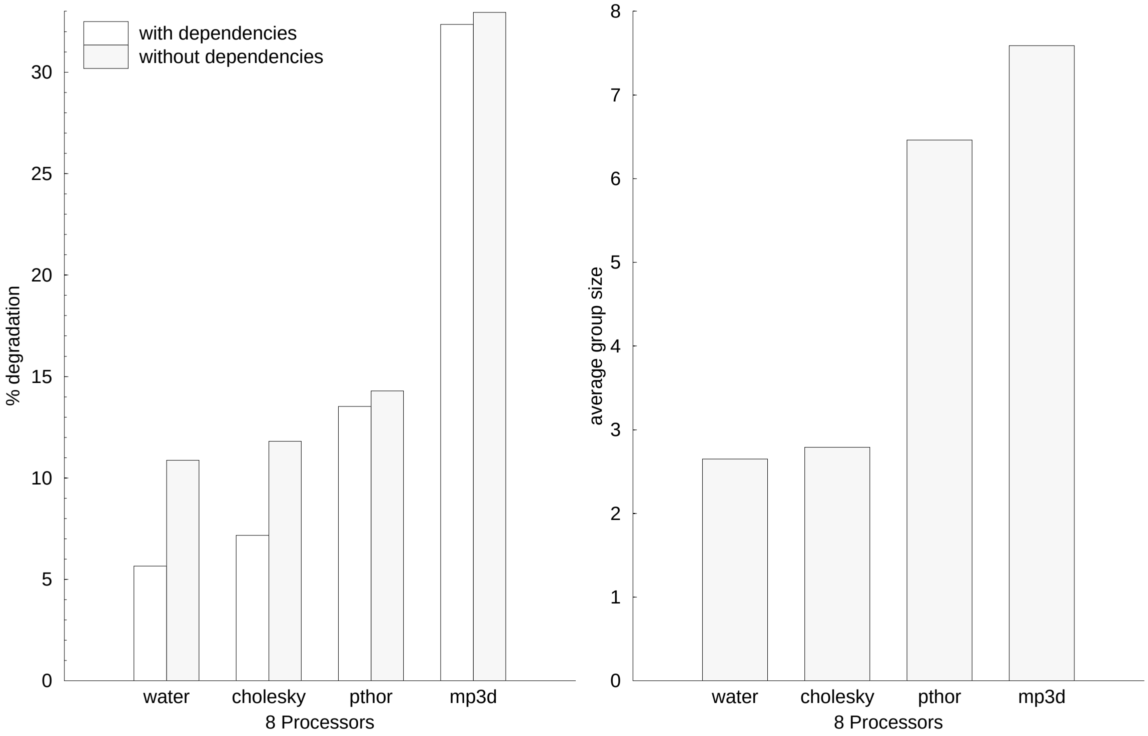


Figure 3.7: efficiency of dependency management

Chapter 4

Stable Memory

4.1 FASST Recovery Protocol[1]

This chapter describes more precisely the FASST recovery protocol and discusses issues in the design of a stable memory (SM), which is the main agent of the protocol (also see [Morin et al 92]).

We assume the FASST architecture as shown in Figure 4.1. A cache is associated with each processor, but we do not need to distinguish between cache levels (e.g. between primary and secondary caches). The SM may be composed of several memory modules. For the purposes of this discussion, we assume that it is not possible to insert SM modules during system operation (i.e. to *hot-insert*), so that to change the memory configuration the machine must be stopped and then restarted in a cold start mode.

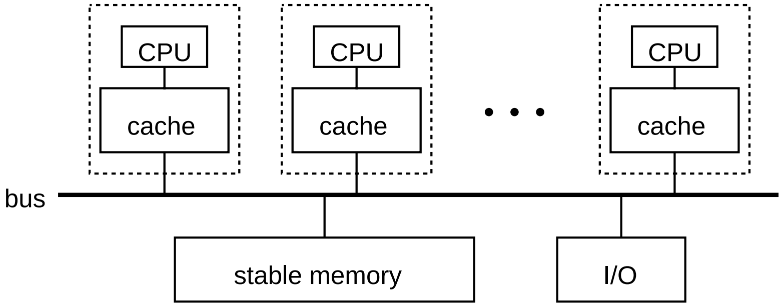


Figure 4.1: The FASST architecture

4.1.1 Stable Memory

Each SM module has the logical structure depicted in Figure 4.2, with two DRAM banks of the same size, $bank1$ and $bank2$, represented by an array of cells [2]:

```
type t_bank = array[0..bank_size-1] of block ;
t_bank Bank1, Bank2 ;
```

In the following, we assume that the SM physical space is divided into a set of contiguous $blocks$ of identical size. Each block consists of a current value in $bank1$ and a recovery value in $bank2$. While read and write commands refer to SM cells, the SM records dependencies on a block granularity.

Various attributes need to be stored with each block, e.g. the processor identifier of the last writer to a block; this information is stored in the *vector* structure, which may be represented algorithmically as a record :

```
type t_vector_elt = record
                  t_owner owner ;
   .... additional information ....
               end ;
type t_vector = array[0..block_nb-1] of t_vector_elt ;
t_vector Vector ;
```

The *owner* field contains either the identity of the active writer to the block (if any) or the nil value.

The SM also maintains a dependency matrix *M* which records dependencies between processors which share memory blocks. This matrix is updated when necessary during read and write operations. M is a $n * n$ Boolean matrix; n being the maximum number of processors in the architecture. A matrix item $M(i, j)$ set to *true* means that processor P_i is dependent on P_j. Once a dependency group is computed by a processor it is stored in each SM module in the *group* field.

In order to optimize bank to bank copy during phase 2 of the commit, the SM maintains a list of blocks modified since the last commit in the *update* list. The ends of the *update* list are pointed to by the *update_ptrs*.

[1] This chapter contributed by : Christine Morin, IRISA/INRIA, Campus universitaire de Beaulieu, F-35042 Rennes cedex France, and Cornelius Frankenfeld, Stollmann GmbH, Hamburg, Germany

[2] For the C dialect used here, a Boolean value *false* is represented by zero, while a Boolean value *true* is represented by a non-zero

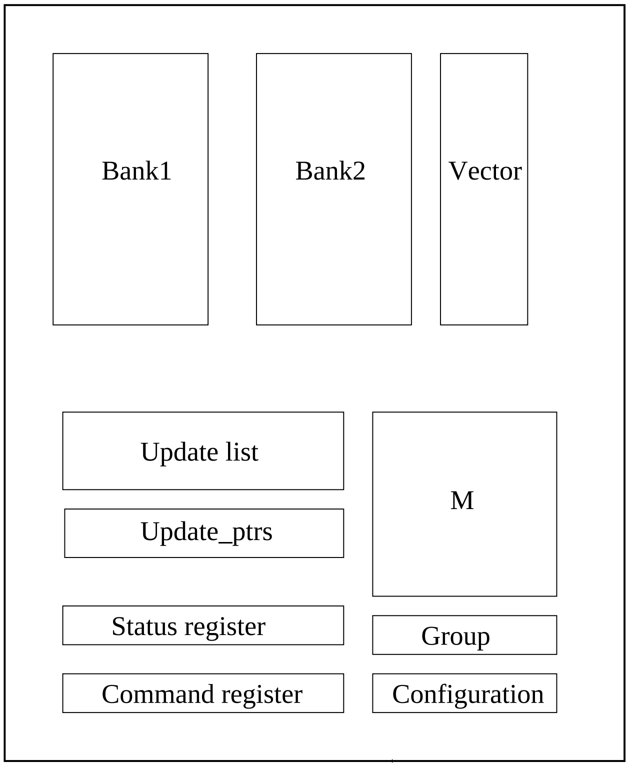


Figure 4.2: Stable memory structure

4.1.2 Dependency management

Data sharing between processors implies processor dependencies. The SM distinguishes between :

(a) read after write dependency, and

(b) write after write dependency.

In the FASST recovery protocol, dependencies are recorded by the SM when processors access memory blocks. The SM also records some dependencies by snooping the cache coherence traffic on the bus. First let us assume that processor P_i is independent from all other processors. When P_i commits then it flushes its cache to $bank1$ and copies all the blocks it has modified since its last commit from $bank1$ to $bank2$. When processor P_i rolls back then all the blocks it has modified since its last commit are copied from $bank2$ to $bank1$ and its cache is invalidated. To detect which blocks have been modified by P_i, each time a block is written to by a processor its identity is recorded with the block in the *owner* field. In other words, the *owner* field associated with a block contains the identity of the last writer to the block. A commit or a rollback of processor P_i implies a bank to bank copy of blocks whose last writer is P_i.

4.1.2.1 Read after write dependency

Consider that processor P_i writes to a memory block *B*. Later processor P_j reads the same block *B* (see Figure 4.3). If the reader of *B*, P_j, commits at time *t* then the writer of *B*, P_i, must also commit dependently with P_j. In fact, if this was not the case then a subsequent rollback of P_i would imply that P_j would have read a value of *B* which was never written, leading to an inconsistent state.

Symmetrically, if the writer of *B*, P_i, rolls back at time *t* then the reader of *B*, P_j, must also rollback dependently with P_i (see Figure 4.4). If P_j does not rollback when P_i rolls back, then P_j possesses a value of *B* which was never written. Because of non-deterministic behavior in a system, nothing guarantees that (after rollback) P_i will write the same value to *B* as it wrote before rollback.

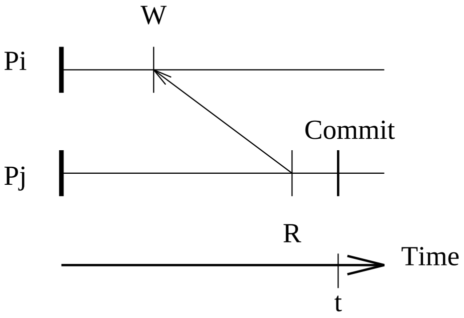


Figure 4.3: Read after write dependency: *Commit*

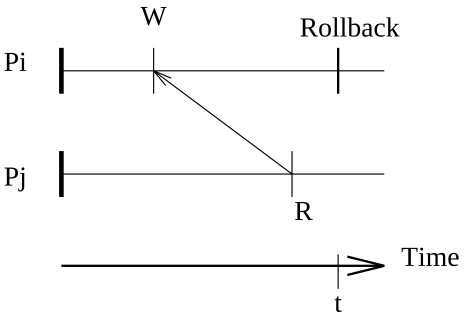


Figure 4.4: Read after write dependency: *Rollback*

In summary, when P_j reads a block previously modified by P_i then P_j is said to be dependent on P_i, i.e. $(P_j \longrightarrow P_i)$. A commit of P_j implies a commit of P_i and a rollback of P_i implies a rollback of P_j.

4.1.2.2 Write after write dependency

Consider that processor P_i writes to a memory block *B*. Later processor P_j writes to the same block *B* (see Figure 4.5). If the first writer, P_i, commits then the second writer, P_j must also commit. The commitment of P_i implies that it is the value written by P_j which is copied from $bank1$ to $bank2$. If P_j does not commit when P_i commits, then a subsequent rollback of P_j implies that *B* is restored to a value which was never written (the value committed by P_i was written to *B* by P_j, which has rolled back).

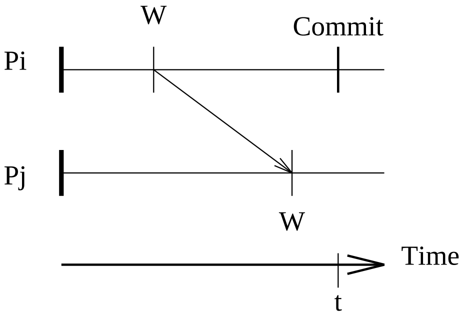


Figure 4.5: Write after write dependency: *Commit*

Symmetrically, if P_j rolls back then P_i must rollback (see Figure 4.6). The rollback of P_j implies that it is the value of block *B* contained in $bank2$ which is restored, which in the general case is different from the one written to by P_i or P_j. If P_i does not rollback when P_j rolls back, then in general P_i possesses a value of *B* which is different to the one it wrote.

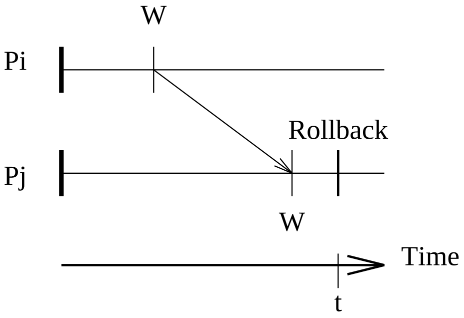


Figure 4.6: Write after write dependency: *Rollback*

In summary, when P_j writes to a block previously modified by P_i then P_i is said to be dependent on P_j, i.e. $(P_i \longrightarrow P_j)$. A commit of P_i implies a commit of P_j and a rollback of P_j implies a rollback of P_i.

4.1.3 Synchronization

In the following algorithms, commits or a rollbacks are called *recovery* operations, and are begun by a processor called the *initiator*. Timeouts are used to guarantee their progress. The following data structures are used:

```
int p_nb ;  /* number of processors                                   */
int sd_nb ; /* number of stable devices, so number of SM modules      */
int active_p[0..p_nb-1] ; /* array indicating which are the active    */
                          /*  (valid) processors                      */
                          /* active[i] == 1 : processor i is active   */
                          /* active[i] == 0 : processor i is inactive */
```

The processor and SM behaviours can be described by two interdependent state automatons.

4.1.3.1 Processor synchronization states

Processor synchronization variables are not stored in the SM. When they are accessed, no dependency is recorded.

```
   /* Synchronization registers - one per processor                   */
type t_p_synchro = array [0 .. p_nb-1] of p_synchro_state ;
t_p_synchro Sync ;

   /* Processor synchronization states                                */
type p_synchro_state = (stopped, stopping, recovery, restart,
atomic_operation, failure, error_handling, waiting, normal) ;
```

A processor may be in the following states (see Figure 4.7):

normal The processor is not involved in a recovery operation, nor has it failed.

stopping This is the state the *initiator* adopts once it starts recovery (commit or rollback). All other processors have to transitate into the *stopped* state. The *initiator* remains in state *stopping* until it knows that all other processors are in the *stopped* state or *timer(2)* expires. Only one processor is allowed to be in this state; this is enforced by a lock operation.

stopped All processors excepting the *initiator* are in the *stopped* state during the computation of the dependency group; they wait in this state until either the bank to bank copy begins or *timer(1)* expires.

recovery The *initiator* remains in *recovery* state until either the SM communicates *copy_done* or *timer(3)* expires. The next normal transition is either to the *atomic operation* state or to the *restart* state.

restart The *initiator* stays in this state until it has restarted all the other processors and has released the lock.

atomic_operation The *initiator* is in this state if an operation is to be atomically executed and committed. It remains in this state until the end of this operation, and then returns to the *recovery* state. If it fails in this state, then a recovery procedure takes place (see Section 4.1.8).

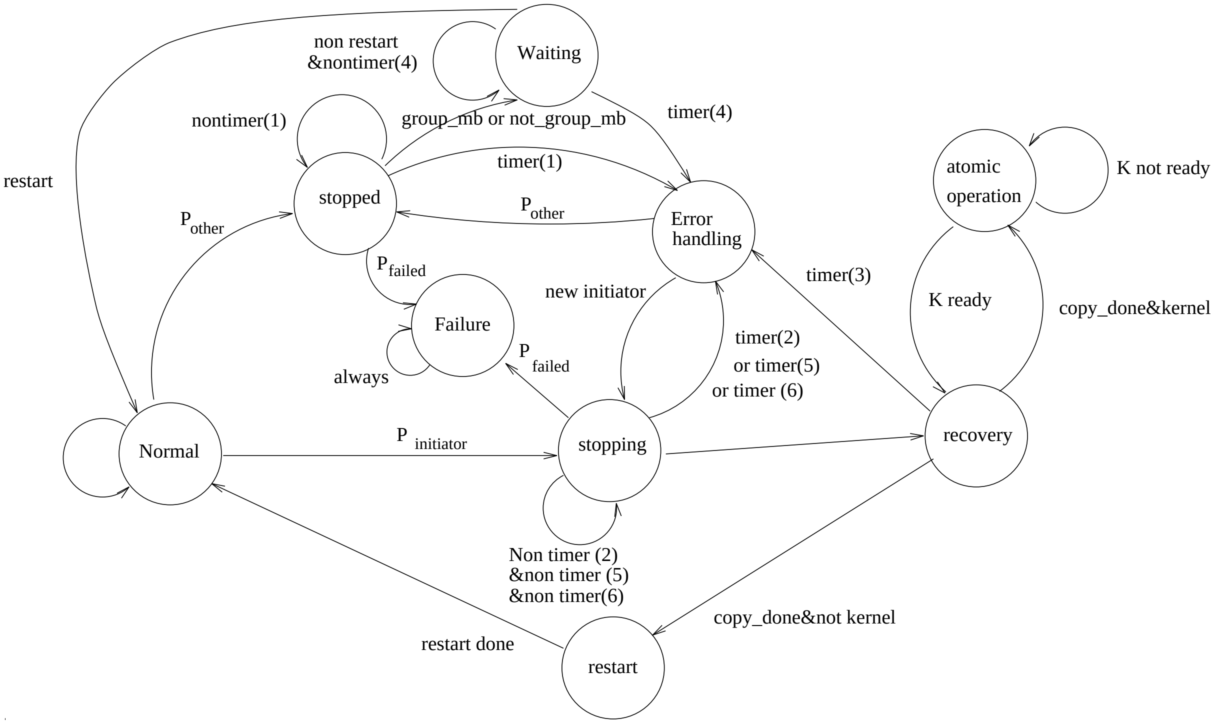


Figure 4.7: Processor automaton

waiting This is the state of any processor, excepting the *initiator*, that is waiting for completion of the bank to bank copy. It waits in this state until it is restarted or *timer(4)* expires.

failure This is the state of a failed processor. Any processor that is in this state is denied access to the bus. It remains in this state until it is physically removed from the system.

error_handling This state is reached by a processor when a timer expires. It may also be reached if some other system failures occur, but these cases are beyond the scope of this book. At least one valid processor enters this state after a processor failure, and normally all of them do. From this state, rollback recovery will always take place. It is possible that more than one processor can fail; one of the failed processors may be the *initiator* itself. In this state, a new *initiator* is chosen, and then the new *initiator* enters the *stopping* state, while all other processors enter the *stopped* state. Once in *stopping* state, the new *initiator* has to compute a rollback group incorporating the newly failed processors.

4.1.3.2 SM synchronization states

Synchronization between SM modules is implemented via status and command registers, which can be represented by a single synchronization variable per module.

```
   /* Synchronization variables - one per stable device       */
type t_sd_synchro = array [0..sd_nb-1] of sd_synchro_state ;
t_sd_synchro sd_state ;

   /* Stable device synchronization states                    */
type sd_synchro_state = (normal, ready, commit_copy,
                         rollback_copy, failure) ;
```

If we assume that the processors are not allowed to restart their execution before the end of the bank to bank copy, this leads to a relatively simple state diagram for the SM (see Figure 4.8). The bank to bank copy (states *commit* and *rollback*) occurs when the *initiator* is in the *recovery* state. The transition between *normal* and *ready* states occurs after reception of the dependency group. The transition from *ready* to *commit* or *rollback* state occurs when the SM receives a *commit* or *rollback* command, respectively.

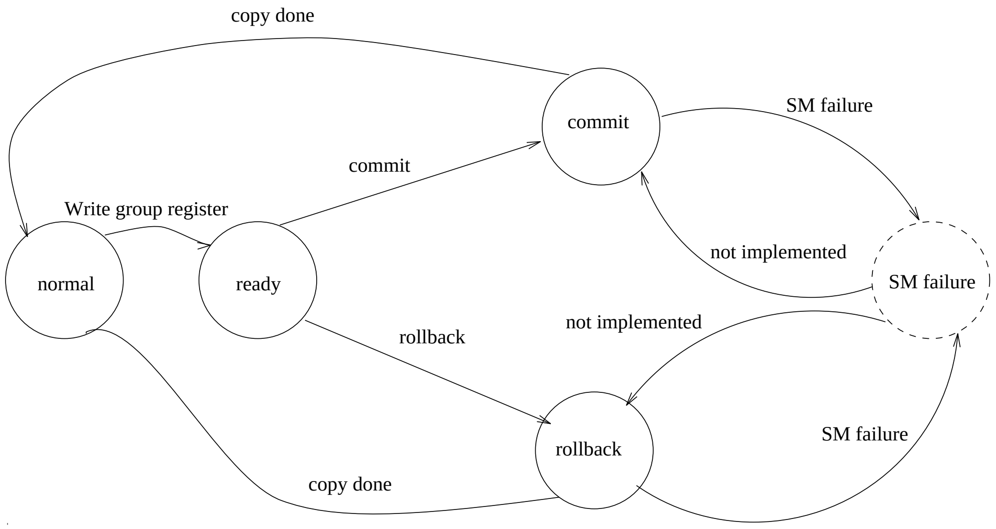


Figure 4.8: SM states

4.1.3.3 Timeout protection

In the following algorithms, the `Wait` primitive is utilized to synchronize concurrent activities. It is is used to stop a processor until the condition given as a parameter is verified by a set of devices or the timer has expired. It is structured as 2 imbricated loops. The internal loop is a loop on the number of devices to be tested (*nb* is the number of processor or of stable devices depending on the condition which is checked). Inside the loop, there is a case on the condition to be tested for each device. As soon as all devices respect the condition, or if the timer expires, the most external loop is terminated.

```
Wait (condition, nb, t_max)
{
   timer_expiration = false ;
   set_timer (t_max) ;
   do
   {
      one_not_ready= false ;
      for (i = 0; i < nb; i++)
         switch (condition)
         {
             case all_stopped :
                /* condition to be tested = Sync [*] == stopped       */
                one_not_ready = ((active_p[i]) &&
                                 (Sync[i] != stopped) &&
                                 (Sync[i] != stopping)) ;
                break ;

             case all_have_flushed :
                /* condition to be tested = Sync[in group] == waiting */
                one_not_ready = ((group & (1<<i))  &&
                                 (Sync [i] != waiting)) ;
                break ;

             case all_sd_ready :
                /* All stable devices must be in the ready state      */
                one_not_ready = (sd_state[i] != ready) ;
                break ;

             case all_sd_copy_done:
                /* All stable devices must be in the normal state     */
                one_not_ready = (sd_state[i] != copy_done) ;
         }
   }
   while ((one_not_ready) && ~(timer_expiration))
   if ~(one_not_ready) unset_timer ;
}
```

Let us examine the different timers used in the protocol (see Figure 4.9).

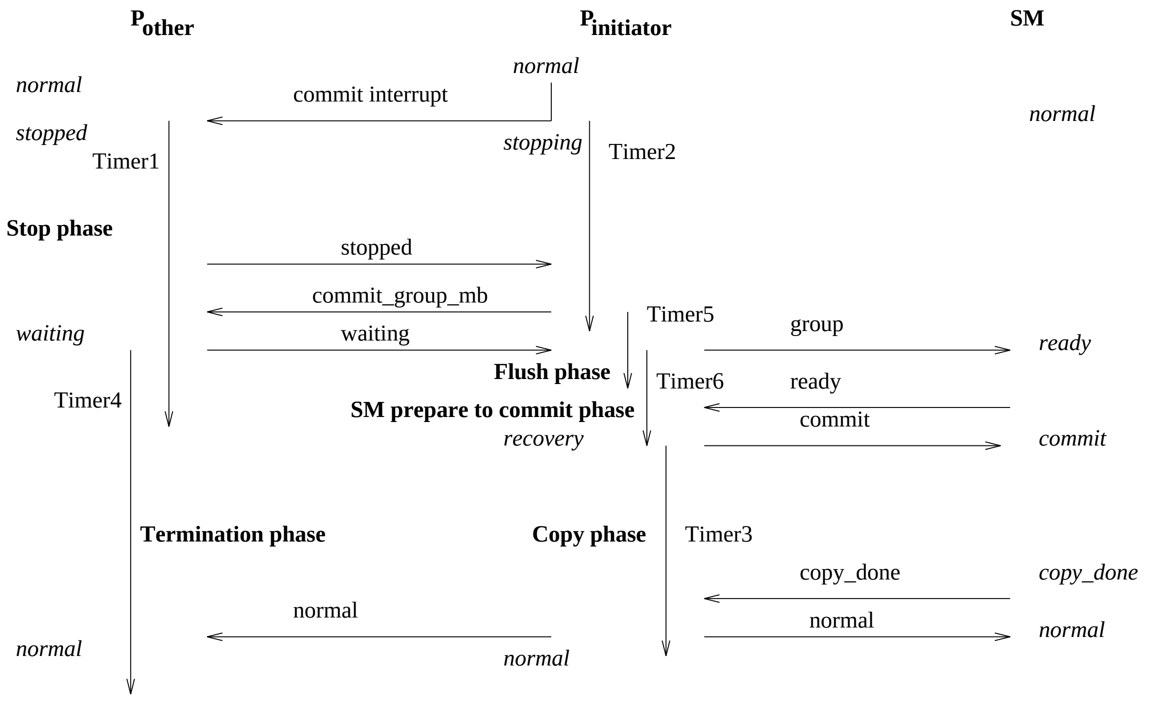


Figure 4.9: Timeout protection

In order to assure fault tolerance, at least two timers must be active during the so-called *stop_phase*. After checking which processors are alive, the *initiator* ”sends” a commit interrupt to all of them and starts *timer(1)*, which covers the maximum duration of this phase. All other processors start their own timers after receiving the commit interrupt. These timers ensure that a failure of the *initiator* can be detected by at least one other processor. The commands *commit_group_mb*, *rollback_group_mb* and *not_group_mb* stop these timers when the *stop_phase* is finished in order to avoid exceptions due to timer expiration. Without these commands, we would have either to deal with exceptions on every other processor, or to use just the timer on the *initiator* (which would not assure fault tolerance).

The expiration of *timer(3)* means that a SM module has failed during the bank to bank copy. Recovery from this requires processor access to $bank2$.

Timer(5) allows the *initiator* to detect the failure of a processor while it is flushing its cache. *Timer(6)* allows the *initiator* to detect the failure of a SM module which does not react to the reception of the dependency group.

Concerning timer durations, *timer(2)* must be longer than any critical section in the system software that is not interruptible by the commit interrupt. *Timer(1)* covers the duration of *timer(2)*. *Timer(3)* covers the duration of the copy phase, and its duration can be calculated by the *initiator* if it knows the top and the bottom pointers of the *update* list. The duration of *timer(4)* has to cover the maximum bank to bank copying time.

Outside these phases, the failure of a processor is detected by some other processor-specific mechanism.

4.1.4 Read and Write Commands

Let us assume that the SM is in *normal* state, i.e. no commit or rollback is in progress, and that a processor is not allowed to issue read and write commands during the bank to bank copy. In the *normal* state, most of the work concerns dependency management. A dependency $i \longrightarrow j$ is noted for:

```
                      (M[i] & (1<<j) == 1)
```

Let us also assume that the conversion between a cell address and a block number is done by the *atb* function, a very basic function implemented in hardware using relevant address lines, and that only line values are exchanged between memory and caches.

Let us further assume that the processor is identified by a *user* field, and that the direction of the transfer is given by *read* or *write* indicators:

```
                      user & read_op <=> reader
                      user & write_op <=> writer
```

4.1.4.1 Read

A read from a cell c will compute the target block b, record a dependency with the active writer of the block (if any) in the matrix M, and will deliver the current value of the cell. Thus:

```
Read (address, reader)
{
   block = atb (address) ;
   owner = Vector[block].owner ;
   if (owner != NIL)
      /* recording a dependency */
      M[reader] |= (1<<owner) ;
   return (bank1[address]) ;
}
```

4.1.4.2 Write

A write to a cell c will compute the target block b, record a dependency with the active writer (if any), change the active writer of the block, and update the current value of the cell within the block b. If this is the first update to the block since the last commit, the block number is appended to the end of the *update* list

```
Write (address, writer, value)
{
   block = atb (address) ;
   owner = Vector[block].owner ;
   if (owner == NIL)
      /* first time this block is modified since the last commit      */
      *update_ptr++ = block ;
   else
      /* recording a dependency */
      M[owner] |= (1<<writer) ;

   Vector[block].owner = writer ;
   Bank1[address] = value ;
}
```

4.1.5 Behaviour of the processor initiating a commit

4.1.5.1 Body of the initiator

We assume in the following algorithm that the *initiator* of the commit protocol is a member of the commit group.

```
Initiate_Commit ()
{
   /* ensure that there is only one initiator,                        */
   /* i.e. only one commit in progress at a time                      */
   Obtain_Commit_Lock () ;

   Sync[my_pid] = stopping ;

   /* Stop all other active processors                                */
   send (commit_interrupt, all_other_processors) ;
   Wait (all_stopped, p_nb, timer2) ;

   /* Every processor is stopped except the initiator                 */
   /* The dependency group is computed by the initiator               */
   Compute_Group (commit, my_pid, group) ;


   /* Inform members of the dependency group that they are            */
   /*   involved in the commit                                        */
   Inform_p (group, commit_group_mb) ;

   WriteRegisters () ;
   Flush_Cache () ;
   Flush_TLB () ;

   /* Wait for all members of dependency group to finish saving       */
   /*   registers and flushing caches                                 */
   Wait(all_have_flushed, p_nb, timer5) ;

   /* Inform non group members to stop timer1 and start timer4        */
   Inform_p (˜group, not_group_mb) ;

   /* Copy dependency group to all stable devices                     */
   Broadcast (group, stable devices) ;

   /* What is important here is that all stable devices commit        */
   /* or none of them does. Problems may arise if the initiator       */
   /* fails while it is requesting the stable devices to commit.      */
   /* Some stable devices may commit while others may rollback,       */
   /* leading to an inconsistent state.                               */

   /* Wait until all stable devices are ready to commit               */
   Wait (all_sd_ready, sd_nb, timer6) ;

   Sync[my_pid] = recovery ;

   /* Commit stable devices                                           */
   Inform_sd (commit) ;

   /* Wait until all stable devices have finished the                 */
   /*    bank to bank copy                                            */
   Wait (all_sd_copy_done, sd_nb, timer3) ;

   Inform_sd (normal) ;

   Sync[my_pid] = restart ;

   /* Restart all processors except the initiator                     */
   Inform_p (((group |˜group)&˜(1<<my_pid)), normal) ;

   /* end of the commit from initiator point of view                  */
   Sync[my_pid] = normal ;
   Release_Commit_Lock () ;
}
```

4.1.5.2 Group computation

When P_i commits then all its descendants according to the dependency relation must commit. The commit group is obtained by computing the transitive closure of the dependency relation. If processor j commits then every k which verifies the following equality has to commit too and recursively:

```
(M[j] & (1 << k)) == 1)
```

Symmetrically, when P_i rolls back then all its ascendants according to the dependency relation must rollback. The rollback group is obtained by computing the transitive closure of the inverse matrix. If processor j rolls back then every k which verifies the following equality has to rollback too and recursively:

```
                      (M[k] & (1 << j)) == 1)
```

The `Compute_Group` primitive forms the global dependency matrix from local dependency matrix located in each SM module and then computes the dependency group *group* related to the processor *p* (given as a parameter). If the *type* parameter equals *commit* or *rollback* this primitive computes the commit or rollback dependency group, respectively.

```
Compute_Group (type, p, group)
{
   /* Computation of the dependency group from the dependency matrix  */
   /* -- Read all matrices from stable devices                        */
   /* -- Build the global matrix                                      */
   /* -- Compute dependency group                                     */

   /* Initialization of M with no dependency                          */
   for (i = 0; i < sd_nb; i++)
      M[i] = (1<<i) ;

   /*  Reading all matrices from stable devices and building          */
   /*    the global matrix                                            */
   for (i = 0; i < sd_nb; i++)
      for (j = 0; j < p_nb; j++)
         M[j] |= Mi[j] ;

   /* Compute dependency group                                        */
   group = (1<<p) ;
   tempo_group = new = group ;
   group_computed = false ;
   do
   {
      for (i = 0; i < p_nb; i++)
         if (new & (1<<i))

            /* i is a new member of dependency group                  */
            for (k = 0; k < p_nb; k++)

               /* looking for dependencies                            */
               switch (type)
               {
                  case commit :
                     if ((M[i] & (1<<k)) &&
                         (tempo_group & (1<<k)) == 0))

                     /* k not already in tempo_group                  */
                     tempo_group |= (1<<k) ;
                     break ;

                  case rollback :
                     if (M[k] & (1<<i)) &&
                        (tempo_group & (1<<k)) == 0))

                     /* k not already in tempo_group                  */
                     tempo_group |= (1<<k) ;
               }

      /* Checking for termination                                     */
      if (group == tempo_group) group_computed = true ;
      new = tempo_group &˜group ;
      group = tempo_group ;
   }
   while (group_computed == false)
}
```

4.1.6 Behaviour of other processors

Let us assume that the rollback interrupt is the highest level (*HL*) interrupt, that the commit interrupt is the (*HL*-1) level interrupt, and that all other interrupts have a lower level. When a commit or rollback takes place the *initiator* sends a *commit_interrupt* or *rollback_interrupt* interrupt, respectively, to all other processors; those processors behave as follows:

```
handling commit_interrupt ()
{
   int old = splx (COMMIT_PRIORITY) ;

   /* Save registers in local scratch                                 */
   Save_Registers () ;

   Sync[my_pid] = stopped ;
   Wait ((Sync[my_pid] == commit_group_mb) ||
         (Sync[my_pid] == not_group_mb), timer1) ;
   switch (Sync[my_pid])
   {
      case commit_group_mb :
         /* The current processor belongs to the dependency group     */
         /* save registers in SM and flush cache                      */
         Write_Registers () ;
         Flush_Cache () ; /* write_back and perhaps invalidate        */
         Flush_TLB () ;
         Sync[my_pid] = waiting ;
         Wait (Sync[my_pid] == normal, timer4) ;
         break ;

      case not_group_mb:
         /* The current processor does not belong to the dependency   */
         /*   group. It waits for the end of the bank to bank copy    */
         Sync[my_pid] = waiting ;
         Wait (Sync[my_pid] == normal, timer4) ;
   }
   splx (old);
}

handling rollback_interrupt ()
{
   int old = splx (ROLLBACK_PRIORITY) ;

   /* Save registers in local scratch                                 */
   Save_Registers () ;

   Sync[my_pid] = stopped ;
   Wait ((Sync[my_pid] == rollback_group_mb) ||
         (Sync[my_pid] == not_group_mb), timer1);
   switch (Sync[my_pid])
   {
      case rollback_group_mb :
         /* The current processor belongs to the dependency group     */
         /* Invalidate cache                                          */
         Invalidate_Cache () ;
         Invalidate_TLB () ;
         Sync[my_pid] = waiting ;
         Wait (Sync[my_pid] == normal, timer4) ;
         /* Read registers from SM                                    */
         Read_registers () ;
         break ;

      case not_group_mb:
         /* The current processor does not belong to the dependency   */
         /*   group. It waits for the end of the bank to bank copy    */
          Sync[my_pid] = waiting ;
         Wait (Sync[my_pid] == normal, timer4) ;
   }
   splx (old) ;
}
```

4.1.7 SM behaviour during recovery operations

```
main ()
{
   /* A recovery operation starts with a write by the initiator       */
   /*   to the group register.                                         */
   Wait (group != NIL) ;

   /* Update dependency matrix                                         */
   /* in order to break dependencies                                   */
   for (i=0; i < sd_nb ; i++)
      for(k=0; k < p_nb; k++)
      {
         if (group & (k<<1)) Mi[k] |= (1<<k) ;
      }
   sd_state = ready ;

   /* The initiator can observe that the bank to bank copy is         */
   /* progressing by the following mechanism. It reads the bottom     */
   /* and top pointers of the update list and by their difference     */
   /* can compute a value for timer(3). A very efficient method       */
   /* is to check if the working pointer is growing                    */

   Wait ((sd_state == commit) || (sd_state == rollback)) ;
   Phase2 (sd_state) ;
   sd_state = copy_done;
   Wait (sd_state == normal) ;
}
```

The `Phase2` procedure consists of the bank to bank copy of blocks whose last writer belongs to the dependency group. If a commit operation takes place the bank to bank copy is done from $bank1$ to $bank2$. If a rollback operation takes place the bank to bank copy is done from $bank2$ to $bank1$. A description of this procedure is given in Section 4.2.6.5.

4.1.8 Atomic operations

Here we propose a mechanism that allows implementation of, for example, critical sections for mutual exclusion (see Figure 4.10). This mechanism requires no specific hardware but allows a good use to be made of the SM functionality.

Such a commit begins like a standard one. When the copy is done the *initiator* can decide if it wants to do an atomic operation (i.e., before restarting other processors) or not; otherwise the standard commit sequence is performed, which ends by restarting all processors. If an atomic operation is to be done, then the other processors are not restarted immediately, but instead the operation is performed by the *initiator*, which then initiates another copy to validate data modified by its atomic operation. At the end of this copy, the other processors are restarted.

If an atomic operation is to be performed then the intention has to be flagged in the first copy operation and to be cleared at the end of the second copy operation, so that a rollback during the execution of the atomic operation can use this flag to uniquely determine the consistent state (K).

4.1.9 Rollback due to a processor failure

Let us assume that processor P_i detects the failure of processor P_k by some mechanism. P_i initiates the rollback of the set of processors which are dependent on P_k. Two situations must be considered:

(1) A commit or a rollback operation is being executed. Consider if a commit operation is already in progress:

(a) If a commit operation is in progress and a rollback operation is triggered before the bank to bank copy phase and before the group has been sent to all SMs, then the dependency matrix is lost, so all the processors must rollback.

(b) If a commit operation is in progress and a rollback operation is triggered before the bank to bank copy phase and after the group has been sent to all SMs, then the dependency group has been computed, so the rollback can be executed (i.e. the commit operation is aborted).

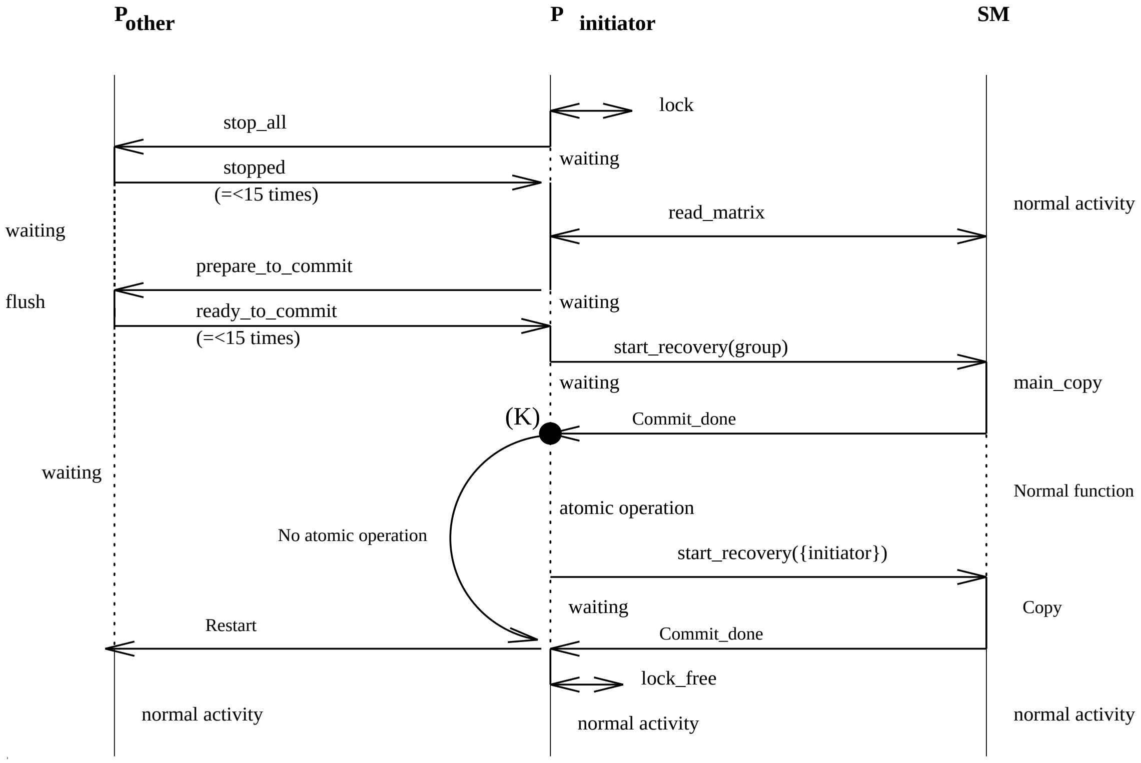


Figure 4.10: Atomic operation

- **(c)** If a commit operation is in progress and a rollback operation is triggered during the bank to bank copy phase, then firstly the bank to bank copy must complete before handling the rollback.

Similarly for rollback when a rollback operation is already in progress.

(2) No recovery operation is in progress. Two approaches are considered below:

- **(a)** Rollback of only those processors that are in the dependency group.
- **(b)** Rollback of all processors; this leads to a simpler algorithm.

These two algorithms are outlined below.

4.1.9.1 Rollback of the dependency group

This algorithm minimizes the number of processors that are *stopped* at the expense of algorithmic complexity.

```
Initiate_Rollback (k)
{
   /* First, check if there is a commit in progress            */
   /* It seems that the commit-lock has to be acquired by the   */
   /* processor initiating a rollback if it is free in order    */
   /* to prevent the beginning of a commit operation (this does */
   /* not appear in the following code)                         */
   if ((test (commit_lock)) != 1)
   {
      /* No commit in progress                                  */
      Initiator = my_pid ;

      Sync[my_pid] = stopping ;

      /* Stop all other active processors                       */
      send (rollback_interrupt, all_other_processors) ;
      Wait (all_stopped, p_nb, timer2) ;

      /* Every processor is stopped except the initiator        */
      /* The dependency group is computed by the initiator      */
      Compute_Group (rollback, k, group) ;

      /* Inform members of the dependency group that they are   */
      /* involved in the rollback                               */
      Inform_p (group, rollback) ;

      /* Check if the current processor is a member of the      */
      /* dependency group                                       */
      if (group & (1<<my_pid)) == 1
      {
         /* Current processor belongs to the group and so must  */
         /* invalidate its cache                                */
         Invalidate_Cache () ;
         Invalidate_TLB () ;
      }

       /* Wait for all members of dependency group to finish    */
       /* cache invalidation                                    */
       Wait(all_have_flushed, p_nb, timer5) ;

      /* copy dependency group to all stable devices            */
      Broadcast (group, stable devices) ;

      /* Wait until all stable devices are ready to rollback    */
      Wait (all_sd_ready, sd_nb, timer6) ;

      Sync[my_pid] = recovery ;

      /* Rollback stable devices                                */
      Inform_sd (rollback) ;

       /* Wait until all stable devices have finished the bank  */
       /* to bank copy                                          */
       Wait (all_sd_copy_done, sd_nb, timer3) ;

      Inform_sd (normal) ;

       Sync[my_pid] = restart ;

      /* Restart all processors except the initiator            */
      Inform_p (((group |˜group)&˜(1<<my_pid)), normal) ;

      /* end of the rollback from initiator point of view       */
      Sync[my_pid] = normal ;
      Release_Commit_Lock () ;
   }
else
   {
      /* A commit is in progress                                */
   }
```

4.1.9.2 Rollback of all processors

Alternatively, if we envisage that rollbacks will be infrequent, we can systematically stop all processors instead of stopping only the processors belonging to the rollback group. This leads to a simpler algorithm, which minimizes the algorithmic complexity at the expense of all processors being *stopped*.

```
Initiate_Rollback (k)
{
    /* First, check if there is a commit in progress            */
   if ((test (commit_lock)) != 1)
   {
      /* No commit in progress                                  */
      Initiator = my_pid ;

      Sync[my_pid] = stopping ;

      /* Stop all other active processors                       */
      send (rollback_interrupt, all_other_processors) ;

      Wait (all_stopped, p_nb,timer2) ;

      /* Every processor is stopped except the initiator        */

      group = all_processors ;

      Inform_p (group, rollback) ;

      /* Current processor belongs to the group and so must     */
      /* invalidate its cache                                   */
      Invalidate_Cache () ;
      Invalidate_TLB () ;

      /* Wait for all processors to finish cache invalidation   */
       Wait(all_have_flushed, p_nb, timer5) ;

      /* copy dependency group to all stable devices            */
      Broadcast (group, stable devices) ;

      /* .....same algorithm as the previous one .....          */
   }
}
```

4.1.10 Various primitives used in the protocol description

The following primitives have been written in order to clarify the presentation of the recovery protocol.

4.1.10.1 Updating and consulting synchronization registers

The `Inform_p` primitive is used by the *initiator* to update the synchronization registers of a set of processors *group* with the value *state*.

```
Inform_p (group, state)
{
   for (i = 0; i < p_nb; i++)
      if ((group & (1<<i)) == 0 ) Sync[i] = state ;
}
```

In the same fashion, the `Inform_sd` primitive is used to update the stable device synchronization variable.

```
Inform_sd (state)
{
   for (i = 0; i < sd_nb; i++)
      sd_state[i] = state;
}
```

4.1.10.2 Locking

Only one commit or rollback is allowed at the same time. This property is ensured by using a global lock.

```
/* Global lock preventing 2 concurrent executions of the commit */
/* or the rollback protocol                                     */
integer commit_lock = 0 ;

Obtain_Commit_Lock ()
{
   while (test_and_set (commit_lock) == 1) ;
   Initiator = my_pid ;
}

Release_Commit_Lock ()
{
   Initiator = NIL ;
   commit_lock == 0 ;
}
```

4.2 Stable Memory Hardware

There are a number of ways to implement a SM. By way of example, we now look at one design that has not been implemented, but is a good illustration of how a SM could be made. The SM module is structured as two identical boards which communicate over a Fast Serial Link (*FSL*), and which are programmed symmetrically as $bank1$ or $bank2$. Each board is organized as 4M x 64bits (32MBytes), protected by 8bits of ECC, using 16Mbit DRAM chips. Table 4.1 estimates complexity, Figure 4.11 shows the board architecture, and Figure 4.12 shows how two of them are interconnected.

4.2.1 Information flow

Figures 4.13, 4.14 and 4.15 illustrate the information flow for the following phases of the commit protocol:

Update The SM is in normal state, dependencies are tracked and the locations of writes to memory are written to the *update* list on the $bank1$ board. The same values are sent to the $bank2$ board over the Fast Serial Link to be written to its *update* list. After this operation, the contents of the *update* lists on both boards are identical.

Commit The group register information is sent to the $bank2$ board, which also starts a commit operation on $bank2$. After this command, packets of block addresses and block data are sent to the $bank2$ board, which compares incoming addresses to its copy in its *update* list. If the comparison is successful, the block data is written to $bank2$ and the packet is acknowledged; otherwise a *NACK* is sent to $bank1$, which repeats the transfer. The *FSL* protocol is responsible for ensuring the integrity of its communications.

Rollback In the case of rollback, the process is identical to that for commit, except that $bank2$ becomes the active sender and $bank1$ compares incoming addresses, writes data, and returns *ACK* or *NACK* to $bank2$. The description of the frame formats is given in Section 4.2.2

4.2.2 Fast Serial Link

Communication between the two boards of a SM module is via the Fast Serial Link, in this case using the *Autobahn II* chip set. These have a data transfer rate of 400 MBytes per second (a whole bank to bank copy would take about 100 milliseconds) over a bidirectional multidrop differential coaxial connection (max.length 50cm) (see Figure 4.16). Alternatively the functionality can be emulated by the hardware shown in Figure 4.17. It is assumed the *FSL* utilizes the 32bit Autobahn frame format (see Figure 4.18), so that any 64bit information which is to be sent over the *FSL* must be split into 32bit words.

4.2.3 Futurebus+ Interface

In this design the system bus is based on the IEEE-896 Standard Futurebus+ [Futurebus+ 94a], and uses a dedicated interface chipset [Texas Instruments 94], for which the following has to be borne in mind:

Partial transactions The use of write-through caches at byte level has to be avoided (uncached data are used for instance for flags).

Split transactions are not supported.

Packet transactions Cache line transfers are possible only as packet transactions. Maximal size of a packet is 8 memory cells (64 bits). Packets have to be aligned on the beginning of a block.

Master property The SM cannot become master of the Futurebus+, and so cannot generate interrupts.

User field There are several ways that the *user* field could be provided over Futurebus+, for example, the upper bits from the 64bit address could be used for this purpose, or some of the tag bits could be used.

4.2.4 Other considerations

The following issues need to be considered:

(a) As noted in 4.1.3.3, the expiration of *timer(3)* means that a SM module has failed during the bank to bank copy, and recovery from this requires processor access to $bank2$. This requires a second Futurebus+ interface.

(b) Given the power requirements for such a large memory capacity, the battery necessary for maintaining power to $bank2$ during mains AC power failures needs to be implemented externally.

(c) In order to increase fault tolerance, the connection to $bank1$ could utilize the Futurebus+ capability to reduce the bus width in the event of a failure of some bus components.

(d) The *vector* memory, *update* list and dependency matrix M are implemented using Static RAMs, and are protected with ECC. The *MATRXCTL* and *COPYCTL* units that control these contain general logic, and are implemented with FPGAs.

4.2.5 Copy-on-write

Copy-on-write is an optimization that allows processors to be restarted before the end of the bank to bank copy. Such a mechanism could be implemented, but first it is necessary to establish whether global performance is improved or not, because:

(a) During the requested memory transfer, it is possible to compare the *owner* field in the *vector* memory with the dependency group information, and if the block belongs to the group, send the block directly to the $bank2$ board. If a commit is in progress, and is interrupted to perform the copy-on-write, then care must be taken that synchronization is not lost between the interrupted address and corresponding data, especially if they are pipelined. On the other hand, it is possible to wait until the commit has completed, at the expense of performance.

(b) In the case of copy-on-write, the operation to copy $bank1$ to $bank2$ has to be interrupted in order to allow the old memory data to be copied even if the *update_ptr* does not reach the corresponding location. If the block has already been copied the new data (from the Futurebus+) can be copied into $bank1$ and new dependency information can be generated. The Futurebus+ operation and the bank to bank copy have to be synchronized, and consequently, some clock periods are lost for every Futurebus+ transfer and for every location copy.

(c) Copy-on-write is only reasonable if the commit time is quite long.

Where	Why	What	Speed	bits/chip	quantity/32MB	I_{cc} mA
$bank1/bank2$	4M locations					
2 x 32MB	64bits data	DRAMs	60nS	4M x 4bits	2 x 16	64/5
	8bits ECC	DRAMs	60nS	4M x 4bits	2 x 2	8/1
vector	256k locations					
2 x 2MB	9bits data	SRAMs	70nS	1Mbits	4 + (4)	70
	7bits ECC	SRAMs	20nS	256kBits	16 + (16)	145
update	256k locations					
2 x 4MB	22bits data	SRAMs	70nS	1Mbits	8 + (8)	70
	7bits ECC	SRAMs	20nS	256kBits	32 + (32)	145

Table 4.1: Assessment of chip count necessary to implement a SM

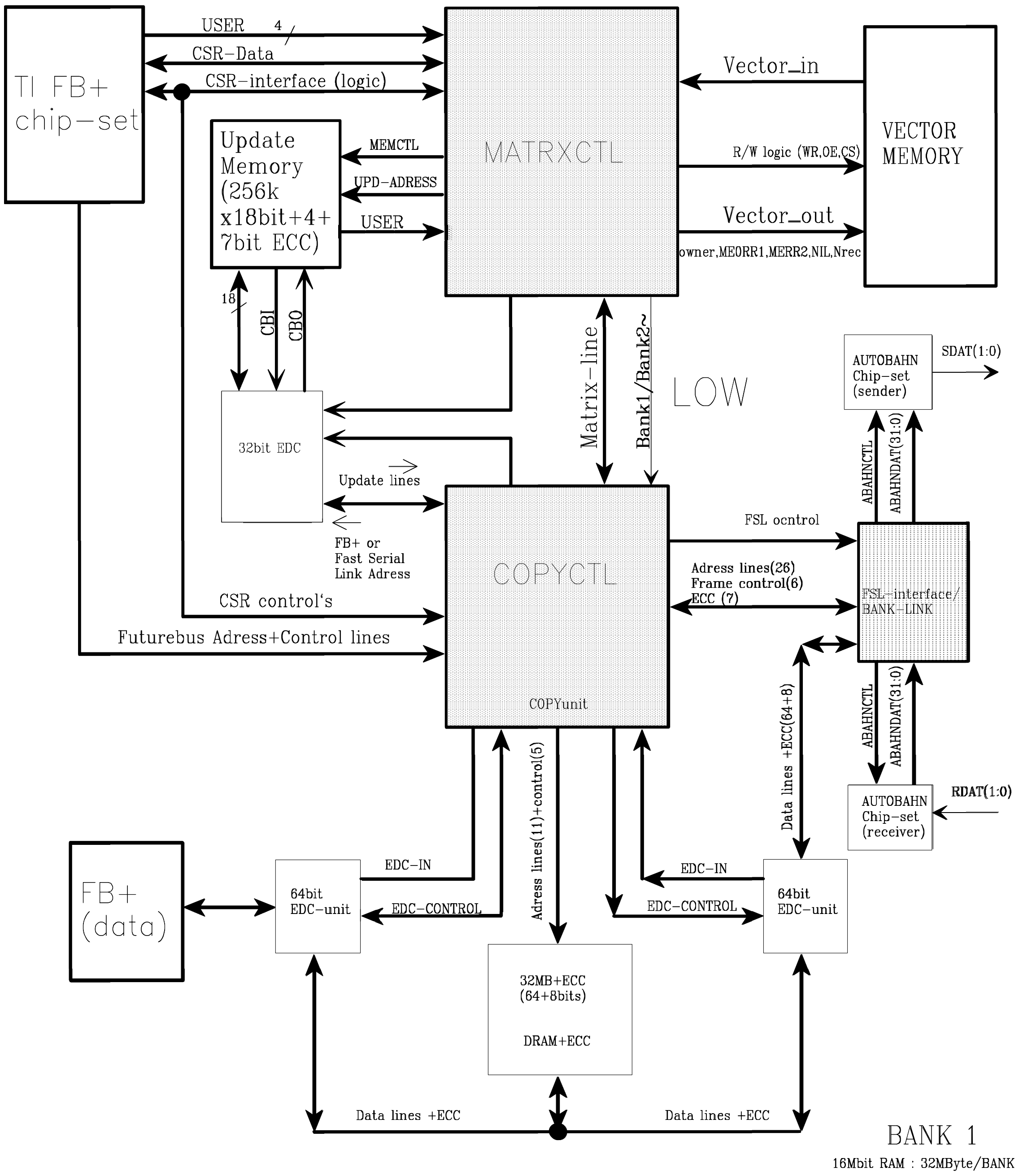


Figure 4.11: SM board block diagram

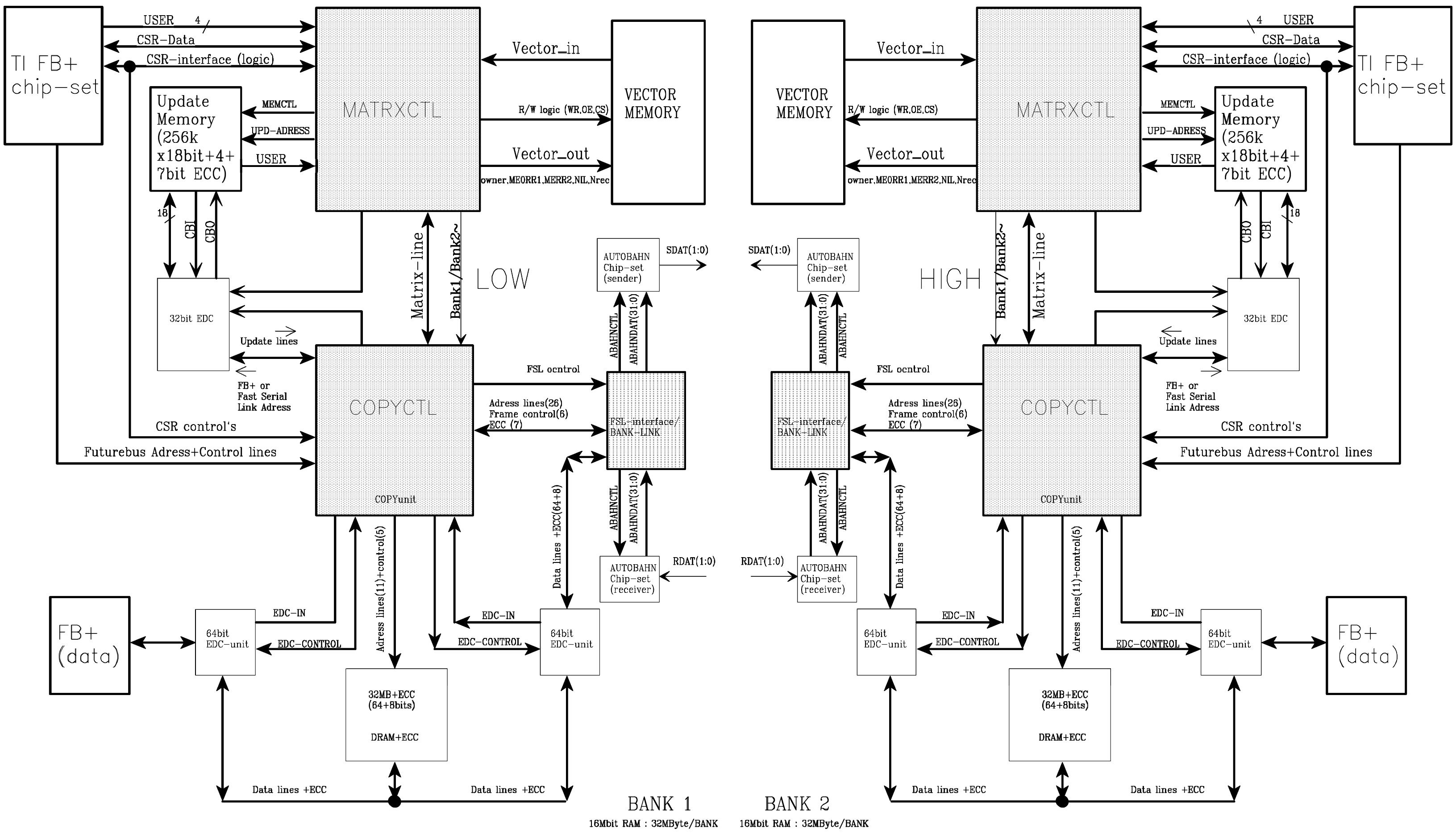


Figure 4.12: Connecting two boards to build an *SM* module

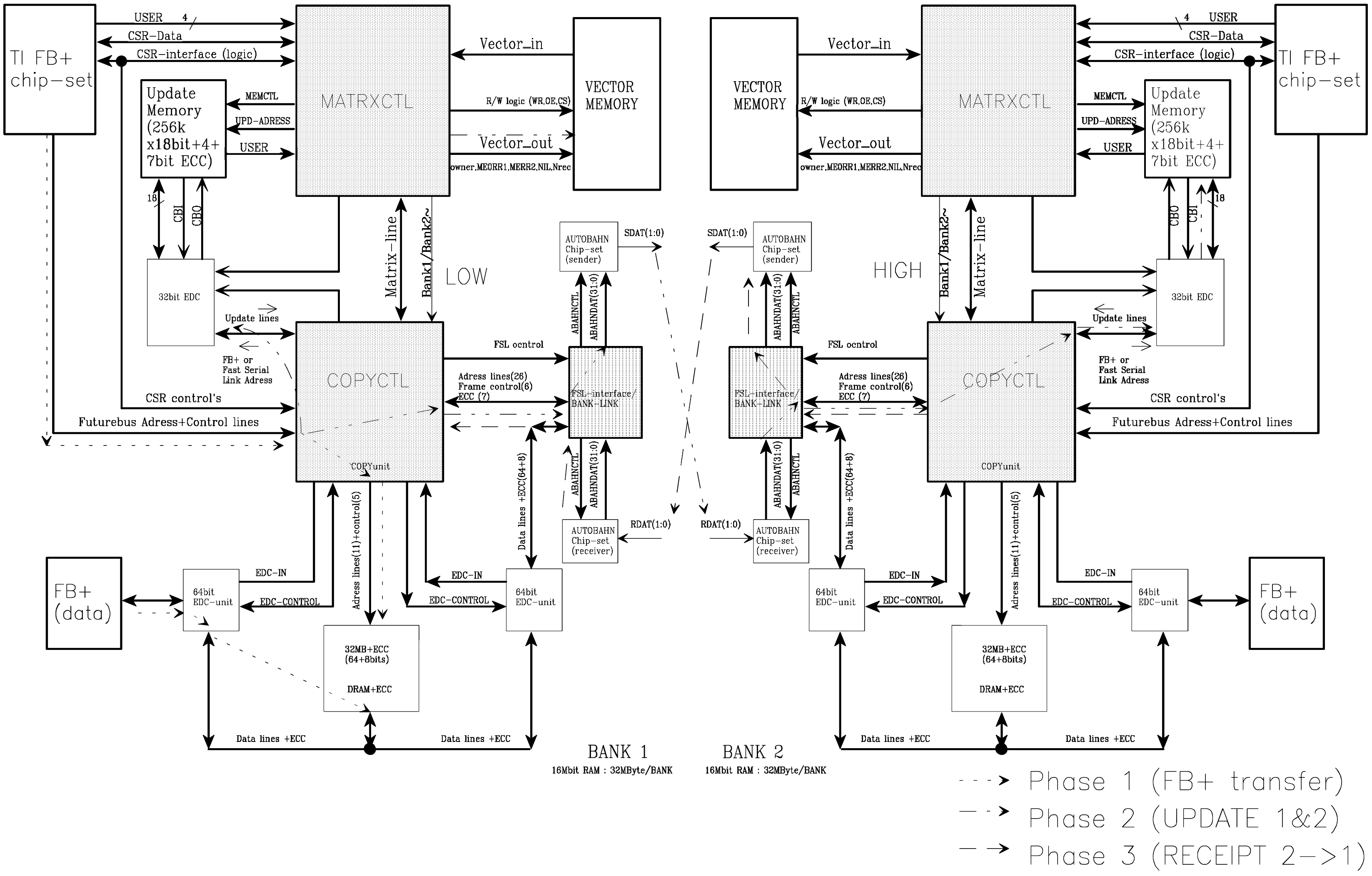


Figure 4.13: Update information flow

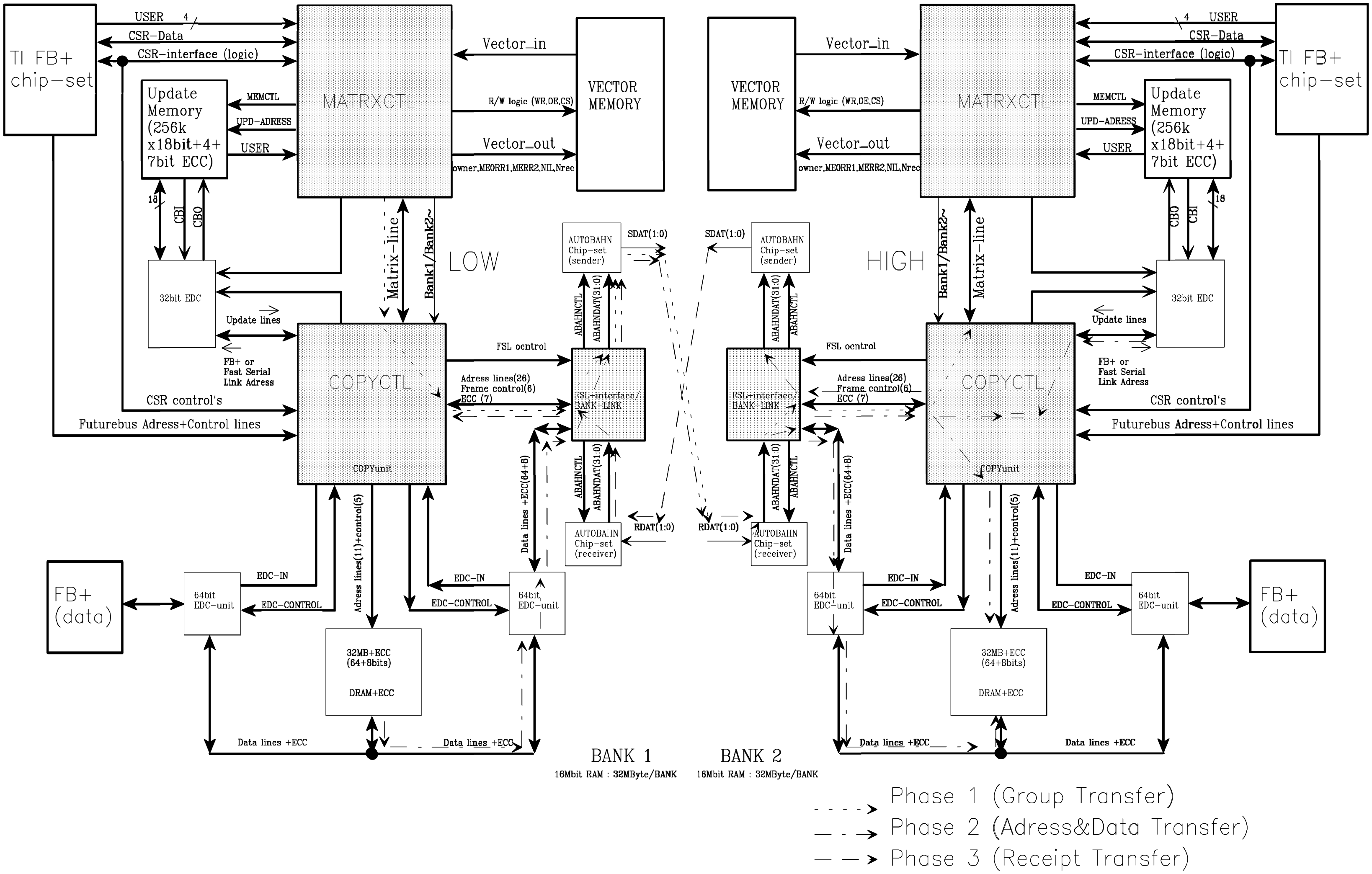


Figure 4.14: Commit information flow

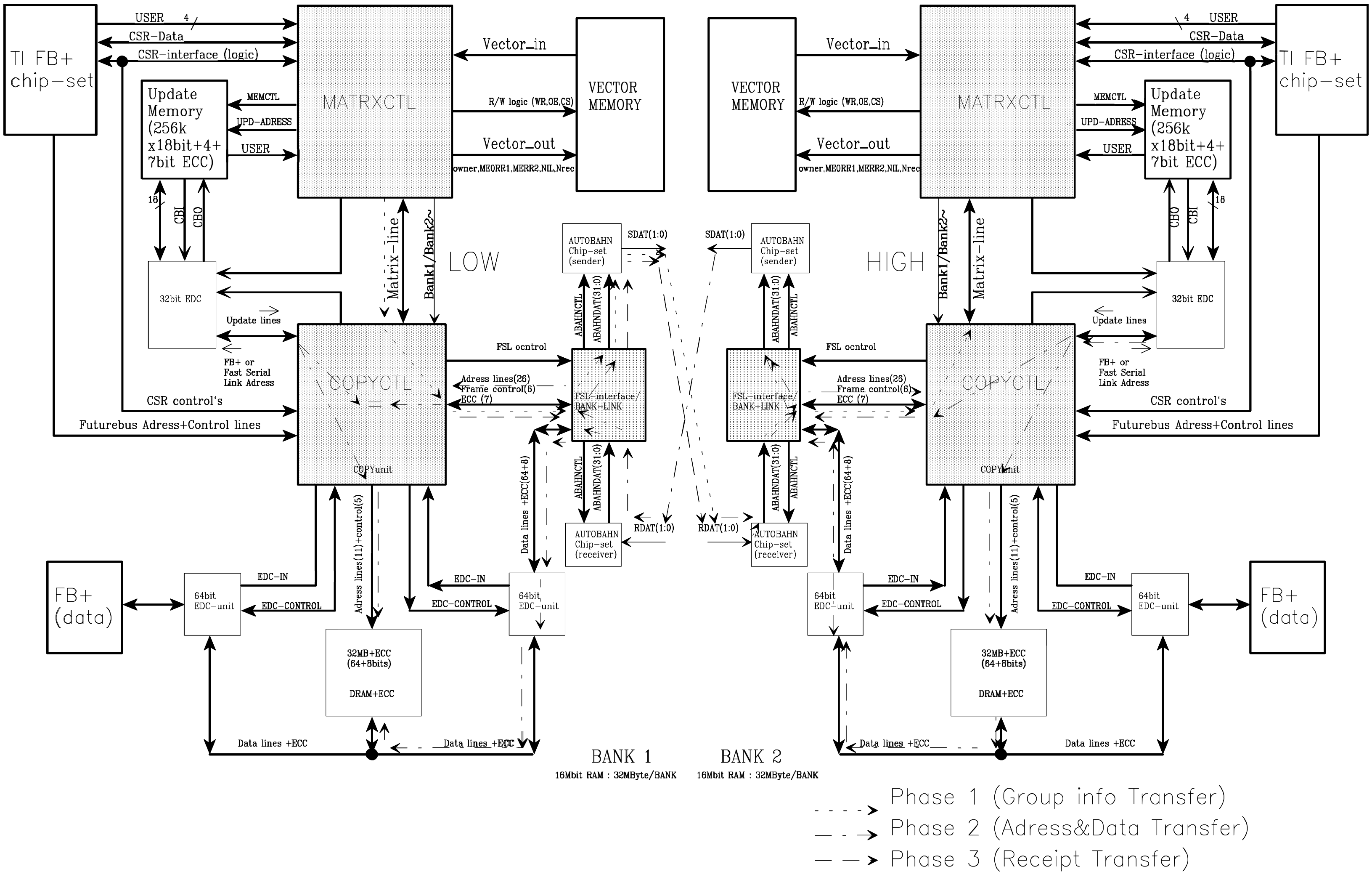


Figure 4.15: Rollback information flow

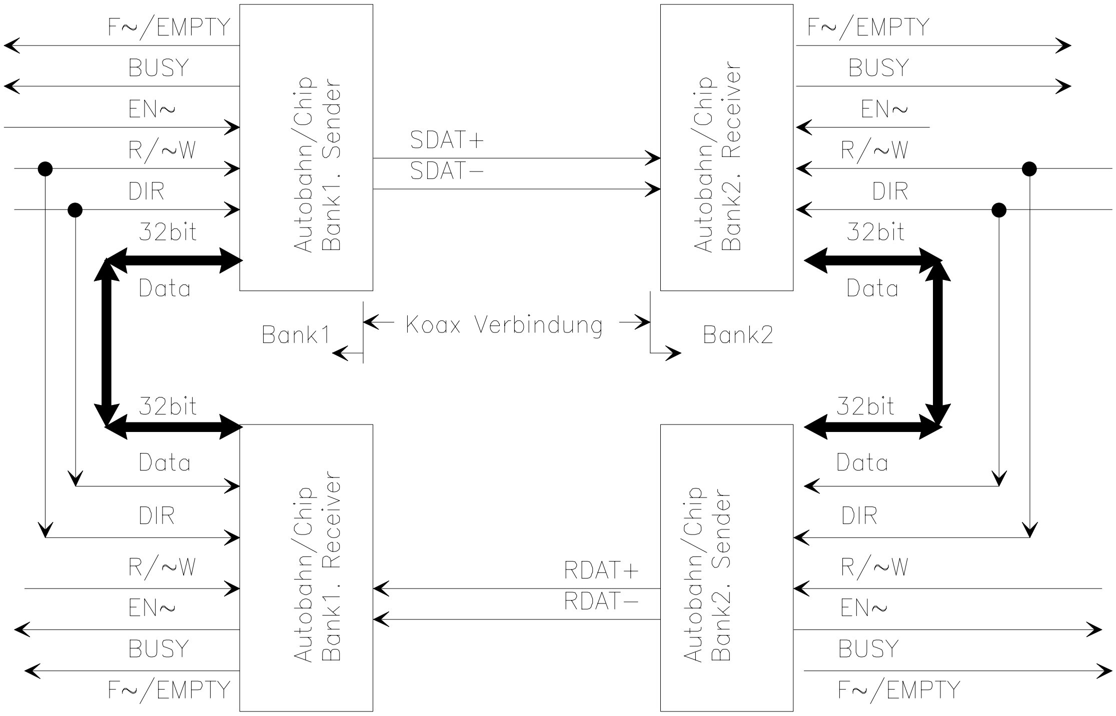


Figure 4.16: Autobahn chipset

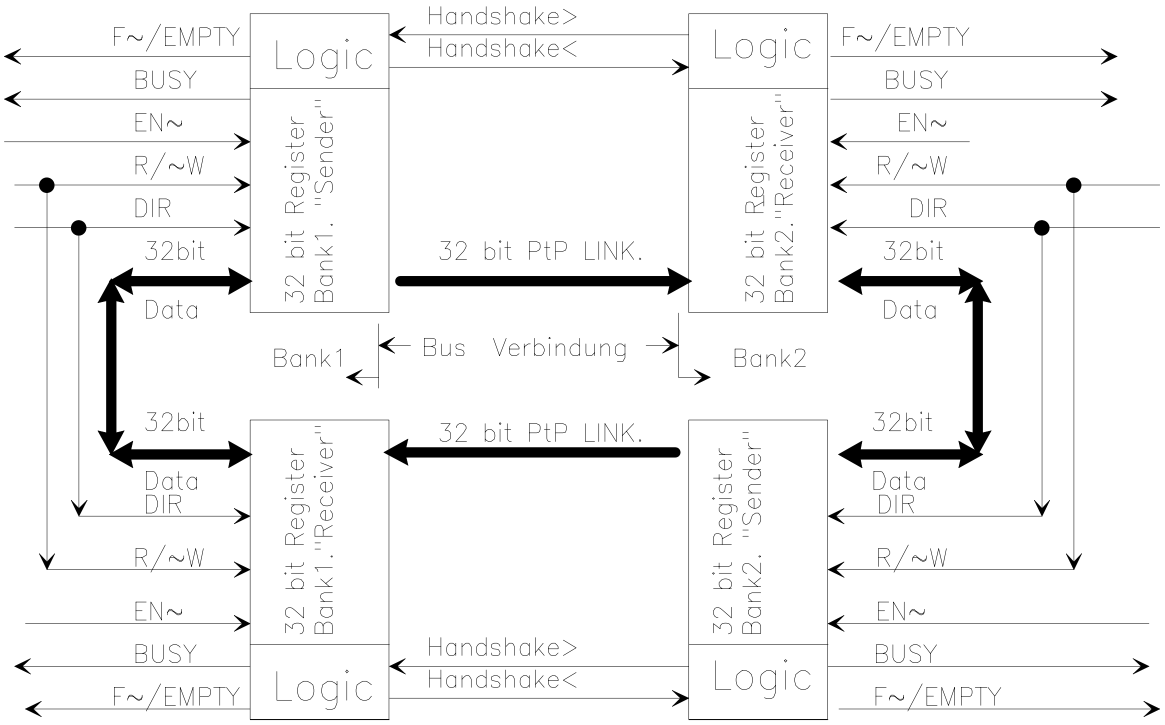


Figure 4.17: Autobahn chipset emulation

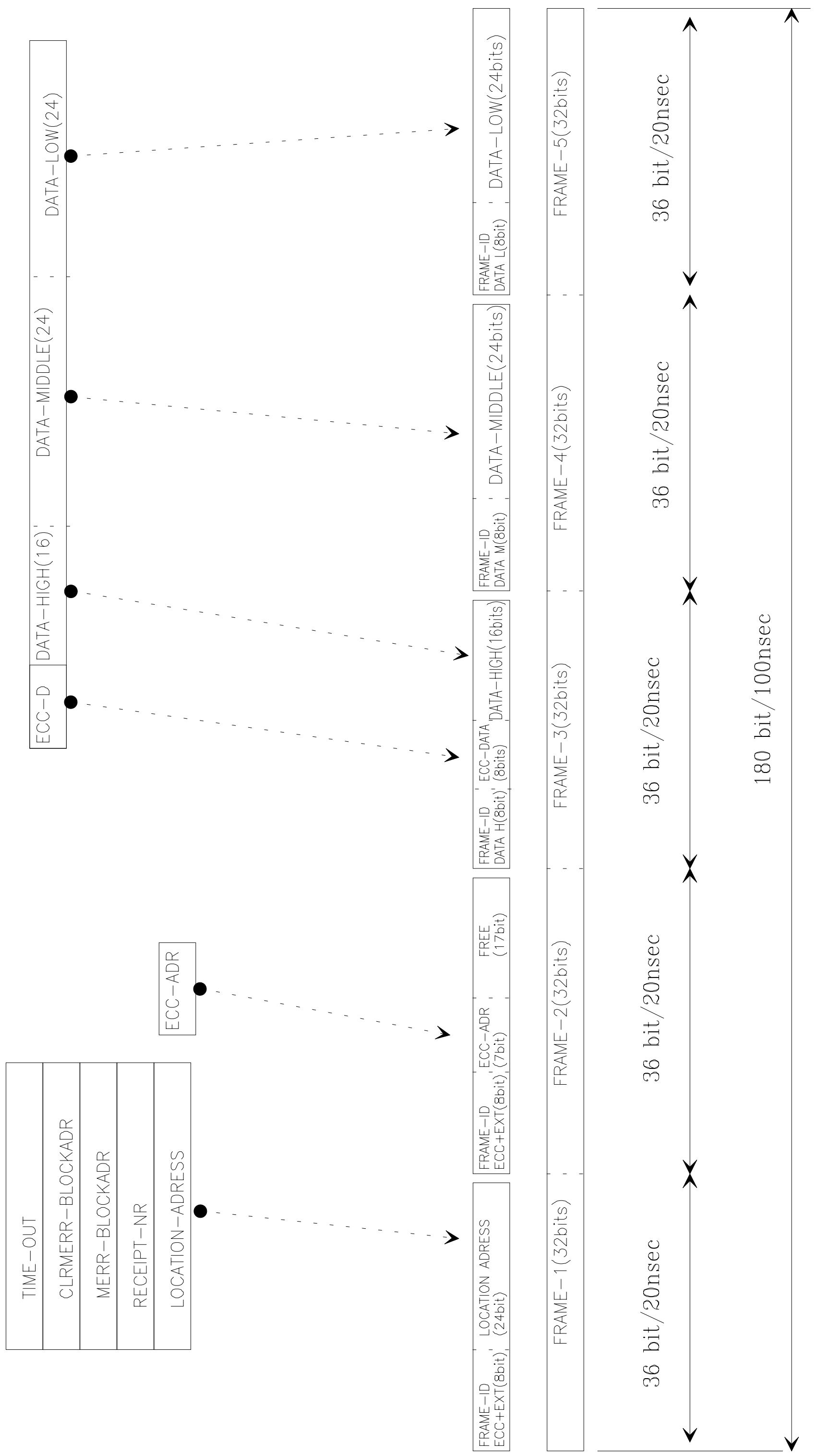


Figure 4.18: Frame format and frames used

4.2.6 System Interface

4.2.6.1 Command and Status Registers (CSR)

Processors interact with the SM via the Futurebus+ CSR area, which is memory mapped. The communication registers are shown in Figure 4.19.

31 21	20	19 16	15 0
reserved	Direction	Initiator	Group information

Command register (write-only)

31 25	24	23 21	20	19 17	16	15 0
reserved	UPD clear	UPD EDCM	Data clear	Data EDCM(2:0)	Warning acknowledge	Start regeneration

Configuration register (write-only)

31	30	29 26	25	24	23	22	21	20	19 16	15 0
VTOR MERR	MTX MERR	MXERRCNT (3:0)	Initialize		MERR		Warnings		Recovery status	Group information
			B1	B2	B1	B2	B1	B2		

Status register (read-only)

Figure 4.19: SM registers

Two EDC chips are used, one for protecting data and the other one to protect the *update* list. The configuration register contains the diagnostic registers for both EDC chips. Three configuration bits, *EDCM(2:0)*, select the EDC mode. A fourth bit, *EDCM(3)*, clears the diagnostic registers.

For recovery purposes, all SM modules involved in the recovery protocol have to deliver their current condition in the status register. *MXERRCNT(3:0)* count the EDC correction operations on the dependency matrix M and *vector* information. Four *MERR* bits indicate multiple bit errors in $bank1$, $bank2$, M and *vector*. Additionally, 4 bits are used for recovery state and two bits (1 per bank) are used to control the initialization process.

The EDC units also allow diagnostic or error information to be read from two 64bit registers, the Error data register and the diagnostic register.

4.2.6.2 Bank addressing

For the SM described, $bank1$ contains 32MBytes of memory. The position of this (its *base address*) within the global address space can be programmed into using a register in the Futurebus+ CSR area. The other SM address spaces are described below.

4.2.6.3 *Vector* memory

The nine active bits of *vector* (see Figure 4.20) are protected by ECC. The information is accessed during dependency tracking via the Futurebus+ transfer address or during copying via the addresses in the *update* list. The processor can address the information of the *vector* memory for read and write using the Futurebus+ two-step indirect addressing; in the first step the location address is written into the indirect address register, while in the second step, data is transferred via the indirect data register (see [Futurebus+ 91] page 42). *MERR B1* and *MERR B2* indicate a multiple bit error in $bank1$ and $bank2$, respectively.

4.2.6.4 Dependency matrix memory

Dependencies are tracked for normal accesses and cache coherency traffic. The dependency matrix memory is organized as an area of 16 x 16 bits (allowing up to 16 processors), protected by ECC, integrated into the *MATRXCTL*

0 3	4	5	6	7	8	9 15
owner	NIL	0	MERR B1	MERR B2	0	ECC

Figure 4.20: *Vector* information

FPGA. The information is updated during the normal activity of the SM, and also can be read by the processor via the CSR area.

4.2.6.5 Update memory

The *update* memory is a single list which records the block number of all modified blocks. A block number is kept in the *update* list until the two corresponding blocks in each bank are made identical by copying. The *update* list management algorithm is a 2 pointer sorting algorithm implemented in hardware:

```
Gap_location = update ;
i = update ;
while (i < update_ptr)
{
   block = *i ;
   owner = Vector[block].owner ;
   if (owner & Group)
   {
      switch on type
      {
         case commit : Bank2[block] = Bank1[block] ; break ;
         case rollback : Bank1[block] = Bank2[block] ;
      }
      Vector[block].owner = NIL ;
   }
   else
   {
      *Gap_location = *i ;
      Gap_location++ ;
   }
   i++ ;
}
```

At the end of the reorganization of the *update* list, the *update_ptr* must be loaded with *Gap_location* value, which is initialized to point to the bottom of the *update* list after power-on.

4.2.6.6 Snooping

The SM must snoop on cache coherency traffic for dependency tracking. This snoop interface is very system-dependent, leading to questions about the compatibility of the cache coherency protocol and the FASST recovery protocol, and about the implications of this protocol for the dependency tracking, that are outside the scope of this book.

By way of example, let us assume that a flush does not invalidate data in cache. Consider a cache line in the *exclusive modified* state of the *Berkeley* protocol, that is subjected to a flush at a $commit$. Consider also that this line is not referenced in the interval to the next $commit$. The line must be invalidated after the flush by some other mechanism, otherwise it will be flushed again at the next $commit$. Issues like this are very system-specific.

4.2.7 Initialization phase

After power-on the system can be in either *cold-init* or *warm-init* state. From the SM point of view, a *cold-init* means that both banks must be cleared, and all previous history (even in $bank2$) discarded, whereas a *warm-init* means that all of $bank2$ must be copied to $bank1$ using the *regenerate* command. At the beginning of the bootstrap the *Monarch* processor (see [Futurebus+ 94a]) must determine by some means whether it is a *cold* or *warm-init*. The system is organized such that an area of memory in $bank2$ contains valid data which allows the *Monarch* to make this decision. As this data is crucial to system integrity, it must be made proof against multiple errors, using N-modular redundancy if needs be. The initialization phase is as follows:

(a) $Bank1$ is first initialized; $bank2$ is not initialized.

(b) The *Monarch* tries to find valid information in $bank2$.

(c) If the *Monarch* does not find valid information in $bank2$, it assumes a *cold-init*.

(d) If the *Monarch* finds valid information then it assumes a *warm-init*. $Bank1$ initialization will take up to 1 second; during this time, the *Monarch* polls a status bit in the status register.

(e) System dependent information in *vector* memory can then be restored from information contained in $bank2$. The $owner$ and *MERR* fields are reset to NIL.

(f) The *Monarch* sends the command *regenerate* to the SM modules and waits for the whole of $bank2$ to be copied to $bank1$. *Vector* and *update* information is ignored. The dependency matrix M is cleared.

(g) At this point, the system is ready to be booted.

As it can be seen above, battery backup is not needed for $bank1$, *vector* or the dependency matrix M. The consistency of data in $bank2$ must be assured by battery backup, and to assure refresh during power-down, the $bank2$ DRAM controller must also be battery-powered. There must also be a sufficient reserve of battery power to accomodate a power failure during any copy of $bank1$ to $bank2$; this requires a *battery-power-fail* signal at least 160mS before battery power fails (new recovery requests which arrive after a *battery-power-fail* is detected may be ignored by all processors and SM modules).

4.3 Fault tolerance issues

The processors request services from the SM and receive results. They can request to read or write data in the SM, or to read SM status, or to start a SM command. The results can be in memory or CSR space. For the processors, a SM command and its results are atomic. It is important to look at the paths over which the information flows (represented by arrows in Figure 4.21). We can distinguish between normal operations, where data is read or written to memory, and status reads or command writes.

Normal activity The processor issues address, direction and size information, and for writes, data, to the Futurebus+. This information flows through the Futurebus+ interface, *COPYCTL* unit, and EDC chip to the DRAM. The resulting read data and/or acknowledgement flows back over the EDC chip and the Futurebus+ interface to the processor. *MATRXCTL* tracks dependencies both for normal accesses and cache coherency traffic.

Failures There is always a finite possibility of a failure of any functional unit, or of the connections between them. The Futurebus+ interface can detect one bit errors on every byte of the address, command or data. The EDC and memory bank can detect two bit errors and correct one bit errors. The *COPYCTL* can compare memory address lines against the address information received from the Futurebus+ interface. *MATRXCTL* counts corrections to errors in M and *vector*, and the counts can been read from the status register. A failure of the *MATRXCTL*, the *COPYCTL* or of the Futurebus+ interface is detected by timeouts of the Futurebus+ interface.

Recovery operation The recovery operation additionally requires error-free operation of the Fast Serial Link.

Notice that a second Futurebus+ interface is necessary to allow recovery from a failed Futurebus+ interface or *COPYCTL* unit.

4.4 Expected performance

The expected performance relates to the services the SM modules deliver. Mostly these depend on the timings achieved by the Futurebus+ interface. Assuming a 50MHz clock for the bus activity, and that bus arbitration is overlapped with data transfers as for the Futurebus+ parallel protocol, then we can expecting the following performance:

Normal memory read or write Here we assume that dependency tracking is performed without loss of performance.

```
write tAA >= 60nsec
     tcyc >= 100nsec.
read  tAA >= 80nsec
     tcyc >= 100nsec.
```

Packetized memory read or write Assume that the packet begins at a block boundary. The cycle time encompasses eight memory line transfers; the first of these needs a full DRAM access, but every subsequent line can use the page mode capability of the DRAMs, taking just 40nS.

```
write  tcyc >= 700 nsec
read   tcyc >= 700 nsec
```

This timing assumes no correction activity is needed. If there are one bit read errors then three additional clock periods need to be added for every corrected memory line. Note that (a) partial transfers are as slow as memory line transfers, because any modified data has to be written back to the memory, taking an extra three clock periods, i.e. 60nS, for every partial write, and (b) if copy-on-write is implemented (see Section 4.2.5) then two synchronisation states (each of 20nS duration) are needed for every memory transfer, so that:

```
tcyc >= 140nsec
```

CSR operations (read or write) The CSR operations are assumed to have:

```
tcyc >= 100nsec for Registers
tcyc >= 200nsec for ROM reads
```

Recovery operation Recovery operations (commit or rollback) are assumed to occur without errors on data or address transfers over the *FSL*, and without ECC errors during data reads from $bank1$ for *commit* or $bank2$ for *rollback*, respectively. Times are calculated for one block. The total time is obtained by multiplying the block time by the number of blocks to be recovered. A block transmission which is *NACK*-ed takes the same time to be retransmitted:

```
tcyc >= 900nsec/block.
```

Initialisation This operation takes place after power-on. We assume that the initialisation uses the page mode of the DRAMs (a page has 16384 memory lines) and takes:

```
tini(page) >= 82usec/Page
tini >= 170msec/bank.
```

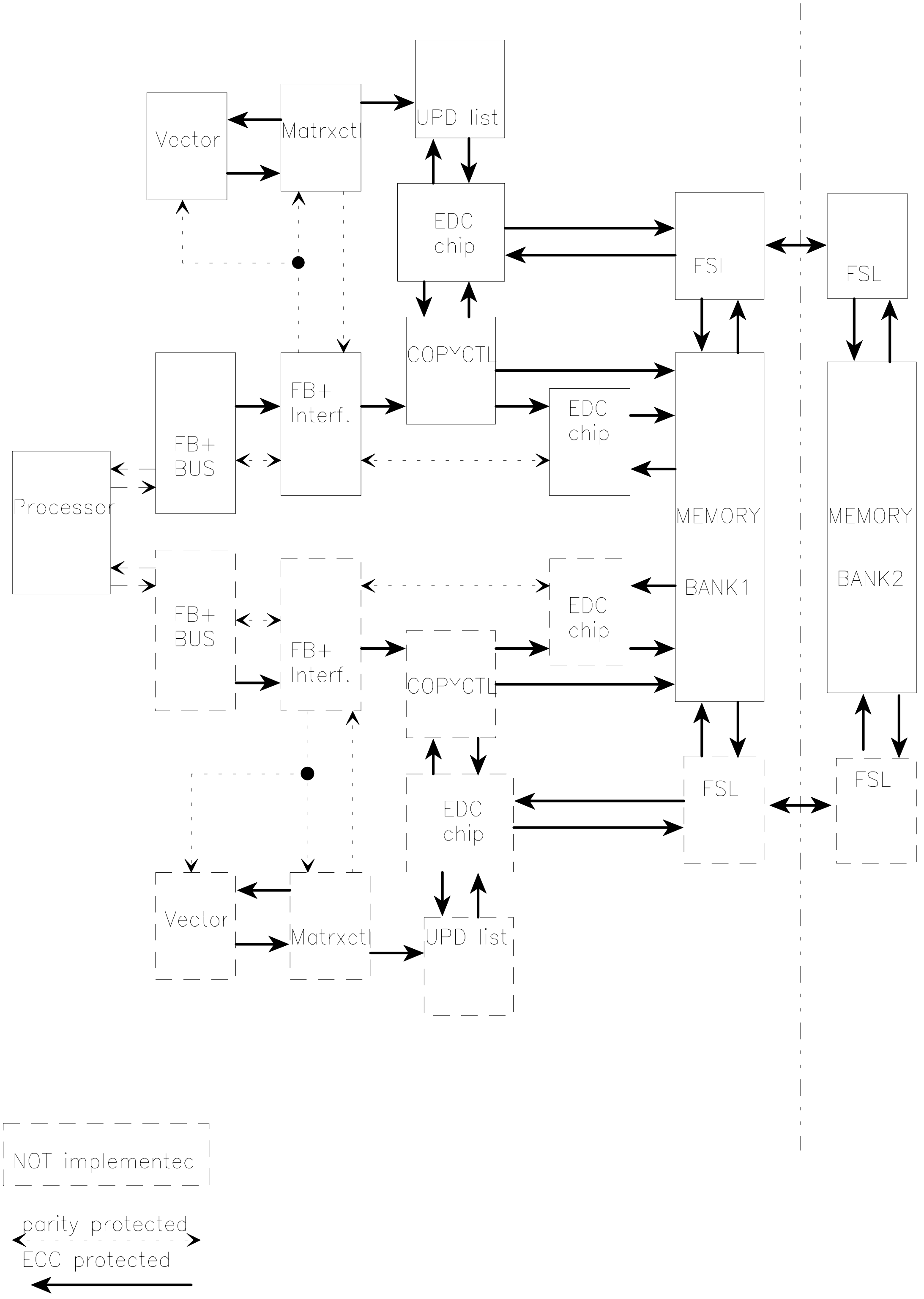


Figure 4.21: SM information flow

Chapter 5

Processing Units

5.1 Fail-stop Processors[1]

During normal system operation, the SM keeps track of dependencies between processing modules, both during normal accesses to the SM, and by snooping cache coherency traffic on the bus. This information is used to establish consistent checkpoints of dependent processors when it is required [Ahmed et al 90, Banâtre et al 90a] as described in the preceding chapters, essentially by flushing processor and cache state to $bank1$ of the stable memory and then copying modified data from $bank1$ to $bank2$. When an error is detected, a rollback is performed in the stable memory by copying the corresponding data back from $bank2$ to $bank1$, and invalidating the information present in the processors caches. If any part of the system suffers a permanent failure, the system can reconfigure itself by gracefully degrading (any processor could be disabled in a similar way) and only the system performance will suffer.

With this approach to fault-tolerance, failed processing modules are not required to take any active part in the error recovery mechanisms. The only requirement imposed on them is not to propagate erroneous data to the system. Taking into account this philosophy, a *fail-stop* or *fail-silent* design [Johnson 84, Laprie et al 90, Schlichting et al 83] is enough to meet the desired degree of fault-tolerance in the system.

This chapter describes an example design of a fail-stop processor module for the FASST architecture, called a *Dual Processing Unit* (DPU). The DPU has been designed for use with a Futurebus+ backplane, as has the example SM design of the last chapter. This bus standard is particularly suitable for fault-tolerant computer systems, and perhaps it is now appropriate to discuss it in greater detail.

5.2 Futurebus+

Futurebus+ is a high performance, versatile backplane bus. Its specification [Futurebus+ 90, Futurebus+ 94a, Futurebus+ 94b] is technology independent to take advantage of future improved technologies. A set of basic configurations, suitable for different applications, have been standardized as *Profiles* [Futurebus+ 91] that are subsets of the options offered by the specification.

Futurebus+ includes two main logical busses, the *data transfer* and the *arbitration* bus, both using asynchronous and distributed protocols.

The data transfer bus width is specified from 32 to 256 bits (in powers of two). The address bus, multiplexed with the data signals, is either 32 or 64 bits wide. There are also parity signals, eight command signals, seven general control signals and seven unspecified tag signals. The information present on each bus signal is relative to the type of transaction and its phase, resulting in a very complete and complex data transfer protocol.

The arbitration bus is composed of seven signals onto which *competition numbers* are issued, plus five additional control signals and parity. An undetermined number of signals should be added if centralized rather than distributed arbitration is used. Also there are five *Geographical Address* signals, a global *Reset* and two signals for a serial bus. Futurebus+ uses the *Geographical Address* signals to identify each module's address on the bus. This allows for a maximum of 31 (one geographical address is reserved and not valid for any module) uniquely identified modules on each bus segment. Any of these modules may be a bridge to another Futurebus+ segment, arbitrarily extending the complexity of the interconnection network.

Futurebus+ uses the IEEE P1212 Control and Status Register (CSR) Architecture [CSR 91a, CSR 91b] to provide a standard means for system reset and configuration, management of nodes with different capabilities and functionalities, and system coordination activity such as messaging and interrupts. Each physical module of a bus can host up to two nodes that are logical entities with their own CSR space in the system memory map. The Futurebus+ memory map has an addressing structure that allows for a hierarchy of busses, modules and nodes when assigning CSR space addresses (see Figure 5.1).

5.2.1 Data Transfer Bus

A Futurebus+ module requests the Data Transfer Bus to perform a system transaction, formed by a request and a response. There are two basic types of transactions:

[1] The following sections contributed by Rafael Martínez and Gregorio Martín, Instituto de Robótica, Universitat de València, Hugo de Moncada, 4 Entlo., 46010 Valencia, España, and Germán Fabregat, Departamento de Informática, Universidad Jaume I, Campus del Penyeta Roja sn, 12071 Castellón, España.

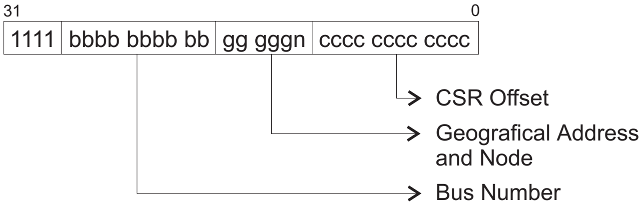


Figure 5.1: CSR Space Address Format

Connected transactions, when the request and the response take place during a unique bus tenure.

Split transactions, when the request and the response take place during two different bus tenures. The responder module has to gain the bus and send the response to the requester.

In the latter case, a bus transaction is composed of two different bus tenures, each of them having the same structure of a connected transaction. Each transaction occupying only one bus tenure (that is, a connected or both halves of a split transaction) have the following structure:

(a) A *connection phase*, when the master selects the desired slave and establishes communication.

(b) An optional *data transfer phase*, where data communication is performed.

(c) A *disconnection phase* when the master ends the transaction by disconnecting from the slaves.

The connection and disconnection phases of the transaction follow an asynchronous handshake protocol. The data transfer phase can be performed in one of the following three ways:

Address only transactions, having no data transfer phase. All the information of the transaction is exchanged during the connection and disconnection phases.

Compelled transactions, where after the connection phase follows one or more data transfers to contiguous addresses. Each data transfer is controlled by an asynchronous protocol.

Packet transactions, where after the connection phase follows a block of data of fixed size. Each individual transfer is synchronous, being the frequency negotiated between the master and the slave during the connection phase.

5.2.2 Arbitration bus and system coordination

The arbitration scheme offered by the Futurebus+ standard is one of the most powerful and versatile included in any existing bus. There is a simple, *central* arbitration scheme where the arbiter receives a pair of request signals for each node capable of being a master, and gives to each of them a bus grant and a preemption signal. Priorities and arbitration policy reside in the central arbiter, although modules can request change of any of its two priority numbers.

There is also a *distributed* arbitration protocol, and the standard specifies the switch from central to distributed arbitration in case the central arbiter fails. In this distributed protocol, each competing node writes its arbitration number into the corresponding signals of the arbitration bus, and a distributed algorithm, involving one or two passes, starts. At the end, only the competing node with the higher number becomes the master elect [Howles 94].

The master elect can be deposed if another arbitration process is started before the tenure takes place, and there is a new competitor with a higher priority number. Also a master can be preempted if it observes that the master elect has a higher priority. These two characteristics make the Futurebus+ specially well suited for the implementation of real time systems.

The arbitration bus and protocols are also used for the sending of messages along the system. Arbitration messages are seven bit numbers, most of them left unspecified by the standard. When distributed arbitration coexists with arbitration messages, the latter have higher priority than arbitration competition numbers. Among the standardized messages are the *Power Fail* (highest priority) and the *Broadcast Interrupts*. The two highest priority messages are not maskable; all others than are maskable by means of the correspond CSR registers.

5.2.3 Cache Coherence

One of the most important features of Futurebus+ as a multiprocessor system bus is the specification of a complete cache coherence protocol, merging the best characteristics of other proposed protocols [Archibald et al 86, Handy 93]. The Futurebus+ protocol is write-back, based on bus snooping. Four states are specified for configuring *MESI* (*Modified*, *Exclusive*, *Share*, *Invalid*) protocols. The attributes of the states are:

Invalid : Any cache line not holding an up to date copy of the system value of the memory line. All lines are placed in the *invalid* state after system reset.

Shared : Any cache line holding an up to date copy of the system value of the memory line, that is also present in other caches of the system.

Exclusive : Any cache line holding an up to date copy of the system value of the memory line, that is consistent with its value in main memory but is not present in any other cache of the system. The module can write to the line privately, changing its state to *modified*.

Modified : Any cache line holding an up to date copy of the system value of the memory line, that is not consistent with its value in main memory (that is, has been written to by the cache's module) and is not present in any other cache of the system. The module is the *owner* of the line, being responsible for responding to transactions requesting it and copying it to memory in the case of replacement.

The specification also defines a set of coherent transactions, that are generated in response to internal activity and cause, either by direct request or by snooping, the state transitions of the protocol. Figure 5.2 depicts the state graph of the protocol. The following coherent transactions are specified as connected transactions:

Read Shared : Non-exclusive read of a line. It is most frequently used to request a line after a read miss. The returned line enters the *shared* or *exclusive* state. Snooping of this transaction forces any existing *exclusive* copy of the line change to the *shared* state.

Read Modified : Read of a line requesting exclusivity. It is most frequently used to request a line after a write miss. The returned line enters the *modified* state. Snooping of this transaction forces invalidation of all valid copies of the line.

Invalidate : Request for invalidation of other copies of a *shared* line. It is used to satisfy a write hit on a *shared* line. Snooping of this transaction forces invalidation of all valid copies of the line.

Copyback : Copy of the line to memory by the module that owns it. The action is mainly caused by a line replacement forced by a miss.

Read Invalid : Read request of a line that is *invalid* before and after the transaction. This transaction is used by I/O devices.

Write Invalid : Copy of a line to memory whose contents have been created by a module not previously having a valid copy of it. Every copy of the line becomes *invalid* after the transaction. It is used by I/O devices.

For the Futurebus+, any snooping protocol that includes the concept of ownership should provide an intervention mechanism to provide valid copies of modified lines. When the *owner* of a (*modified*) line recognizes a transaction requesting that line it must inhibit the memory from supplying the information, and respond to the transaction by providing the requested line.

In addition, and to obtain higher performance, a *snarfing* mechanism is specified. When a module waiting to gain the bus to satisfy a read miss recognizes a non-exclusive transaction involving the missing line, it is allowed to load the line on the fly and then retire the pending request. *Snarfing* is indicated by the assertion of a special bus signal to prevent the snarfed line from appearing as *exclusive* in the (possible) requesting cache.

5.2.4 Reliability

Futurebus+ specification pays special attention to the reliability of the systems. The specification defines two levels of application of measures to improve system reliability. Firstly, a general policy is given for error detection and management, to which standard compliant systems must adhere, and secondly, general recommendations and

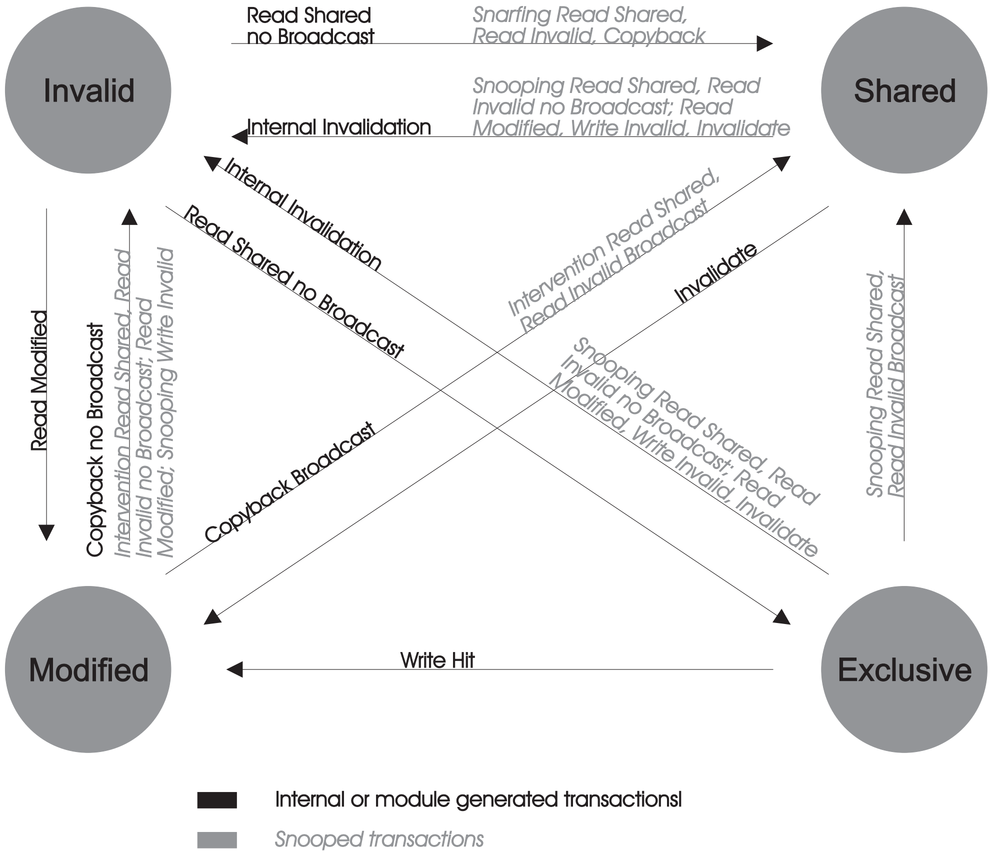

Invalid
Shared
Modified
Exclusive
Read Shared no Broadcast
Snarfing Read Shared, Read Invalid, Copyback
Internal Invalidation
Snooping Read Shared, Read Invalid no Broadcast; Read Modified, Write Invalid, Invalidate
Internal Invalidation
Read Shared no Broadcast
Intervention Read Shared, Read Invalid Broadcast
Invalidate
Snooping Read Shared, Read Invalid no Broadcast; Read Modified, Write Invalid, Invalidate
Read Modified
Copyback no Broadcast
Intervention Read Shared, Read Invalid no Broadcast; Read Modified; Snooping Write Invalid
Copyback Broadcast
Snooping Read Shared, Read Invalid Broadcast
Write Hit
Internal or module generated transactions!
Snooped transactions

Figure 5.2: Futurebus+ Cache Coherence Protocol

preferred options are stated in the specification wherever reliability critical issues are presented. In addition, the documentation of the standard includes specific examples for fault-tolerant systems.

Maintainability is a key point in Futurebus+. Mechanical and electrical specifications are provided for live insertion and withdrawal of modules. In addition, a hierarchy of tests and verifications is detailed, establishing the minimum coverage requirements for each type of fault for all levels. The main objectives of this system of testing are unequivocal detection of faulty nodes, high coverage of system faults, and isolation of failures in the replaceable units of each node.

One of the main requirements is a Built In Self Test (*BIST*) capability on each node. The basic *BIST* and the extensions recommended by the specification are organized according to the following hierarchy:

Initialization Tests activated after system or node reset, to be built in each node and to be capable of unambiguously finding errors in it.

Extended Tests needing a buffer in system memory. These require a node that is capable of becoming a bus master, and detect errors in the interface to the Futurebus+.

System Tests needing a buffer in system memory, and the cooperation of two nodes. These are used to detect global system functions such as parallel protocol, arbitration, etc.

Manual Intervention Tests requiring the intervention of a human operator, generally used to verify external interfaces and devices (I/O subsystem).

These tests include a set of executives, classified according to their characteristics and duration. A very useful distinction is between *default tests* that last less than 10 seconds, and *full tests* without specified time limit. The interface between the tests and the system is a set of CSR registers that used to identify the test capabilities of the nodes, to activate certain tests when written to, and to hold the results of those tests that have been performed.

Error management is also performed in a hierarchical, layered structure. When any level of the specification detects an error, it tries to resolve it at the same level, and only when this is not possible does it store the information about the error in the standard CSR registers, and send an exception to the upper level.

Futurebus+ is very well suited to fault-tolerant systems. Every Futurebus+ protocol includes basic means for error detection. In addition, every set of bus signals is protected with transversal parity, and watchdog timers are included to ensure the progress of the protocol phases. Furthermore, arbitration, cache coherence and message passing are all distributed protocols, which are not affected by the failure of any of the nodes. In addition, a great level of detail of the specification is dedicated to bus bridges (allowing for the implementation of duplicated bus systems), to fault-tolerant power supplies, and to allowing for several levels of live insertion and withdrawal of modules.

5.3 Dual Processing Units

The Dual Processing Unit (DPU) [Martínez et al 95] is the execution engine of the FASST architecture, but the example design outlined below has also been designed as a research vehicle that had to satisfy other constraints:

(a) The system's hardware had to offer, without conflicting with the FASST philosophy, the maximum facilities to build an open fault tolerant multiprocessor.

(b) It had to be designed for future upgrading.

(c) Current standards had to be used where they did not interfere with (a) or (b).

In relation to the FASST architecture, the most important decisions regarding the design of the DPU were:

(a) The DPU had to be a fail-stop processing unit, based on two Intel i486/DX processors running in lock step, for all the obvious reasons.

(b) The system had to be designed around the IEEE Standard P896 Futurebus+ bus, given its suitability to the task.

(c) The Futurebus+ interface had to be implemented using a dedicated Texas Instruments chipset [Texas Instruments 94]. This was a pragmatic choice.

At the time, the Texas Instruments Futurebus+ chipset was the most complete chipset available, but still did not offer the full functionality required for the system. As a result of this, although the design follows the Futurebus+ standard, it does not comply with any of the standard profiles, although almost every feature of profile B is supported except central arbitration and a standard cache coherence protocol.

The system is able to support up to 31 Futurebus+ modules, and the same number of nodes (the Texas Instruments chipset does not allow for more than one node per module). These modules can be of any type with the only restriction that at least one of them be capable of arbitration. The normal configuration of a complete system would be a number of DPU modules, all of them capable of arbitration, a number of memory modules without arbitration capability, and a number of I/O modules capable of arbitration.

In general, as it will be explained later, a module needs to be capable of arbitration in order to be able to send interrupts. A global interrupt scheme provides a common mechanism to broadcast interrupts to the system, or to send interrupts only to individual modules, i.e. to a particular target. Targeted interrupts are supported using the standard CSR registers and both the normal data bus and the arbitration bus. Interrupts can be broadcast using arbitration messages (targeted interrupt messages) or sent to individual modules by accessing them in a normal write to the appropriate CSR register. This feature is fully supported by the Texas Instruments chipset.

Apart from stable devices like the SM, the only other devices the FASST architecture requires to be specifically designed for fault-tolerance are the DPUs. Nevertheless, The Texas Instruments chipset provides parity protection to all local memory devices and local module busses, is able to detect longitudinal parity errors in Futurebus+ transactions and to retry any faulty transactions, and allows for level 2 live insertion-withdrawal as defined in the Futurebus+ standard.

The DPU is designed to allow local or global memory checkpointing, to implement a suitable backward error recovery scheme. As a small extension, the local resources of a failed DPU module may be read by another processor, assuming the fault is confined to the DPU's processor/cache subsystem.

5.3.1 System Memory Map

The system has a 32 bit address physical memory map, thus allowing for a physical global memory of up to 4GBytes. The Futurebus+ standard defines an area of 256MBytes, at the top of the addressable memory, for CSR use, leaving 3840MBytes for System Memory. This memory is distributed into three logical memory spaces in the System Memory; the *Local Private Memory*, the *Local Shared Memory* and the *Global Shared Memory* (see Figure 5.3):

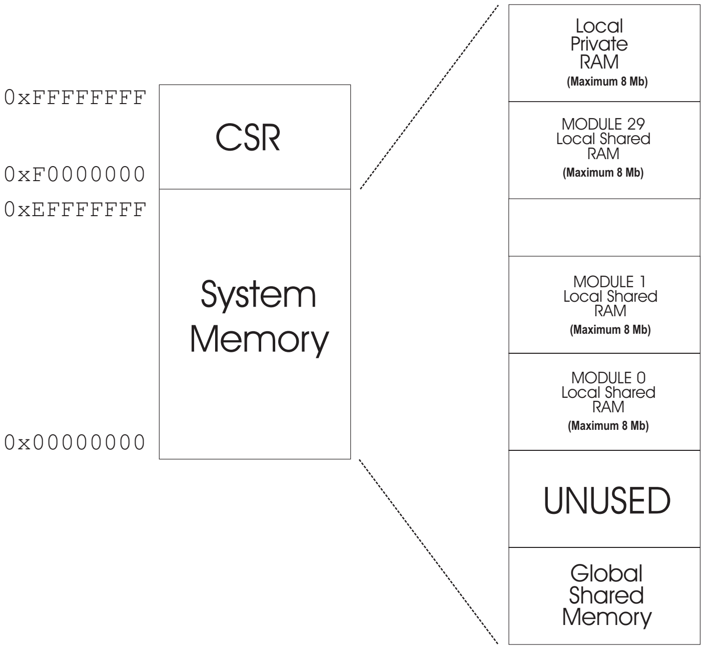


Figure 5.3: DPU System Memory Map

Global Shared Memory (*GSM*) This space is dedicated for memory modules or I/O modules that do not need to be capable of arbitration. It is an external address for any Futurebus+ master, and is located from the lowest address upwards.

Local Shared Memory (*LSM*) This space is dedicated for local memory of modules that are capable of arbitration, and should be globally accessible. The maximum amount allowed is 8MBytes per module, for a maximum of 31 modules (assuming there is no *GSM*). *LSM* addresses are dependent on the *Geographical Address* of the module they belong to, and are located from just below the *LPM* downwards.

Local Private Memory (*LPM*) This space is dedicated for local, private memory such as for start-up and diagnostics code, and should not be accessible outside a module. It is allocated a maximum of 8MBytes per module, all sharing the same global address space equal to the *LSM* space corresponding to the non-existent module 31 (remember, module 31 does not exist in so that its address can be used for CSR broadcast accesses). This space is located from just below the CSR reserved addresses downwards.

With this scheme, and mapping the *LSM* for each module's local memory and the *LPM* for local extended units using the standard CSR registers, most of the on- and off-module transaction address decoding is performed by the Texas Instruments chipset, making the local decoding logic extremely simple.

5.3.2 Interrupt scheme

Interrupts are the most common mechanism of inter-module event signalling. The DPU supports *Unit Specific* and *Unit Broadcast* interrupts as defined in Futurebus+ profile B [Futurebus+ 91]. The interrupt scheme is based on two standardized CSR registers: the *INTERRUPT_TARGET* and the *INTERRUPT_MASK*. These two registers allow for 32 prioritised interrupts, the highest order bit flagging the highest priority, and so forth. *Unit Specific* interrupts are set by a normal Futurebus+ write to the *INTERRUPT_TARGET* register in the selected module. The contents of this register are bitwise ANDed with the *INTERRUPT_MASK* register, and if the result is nonzero then the local module *INT** signal is activated to indicate an interrupt is pending.

Unit Broadcast interrupts are set by using any of the 32 arbitration messages (0x80 to 0x9F) related with the interrupt bits, which set the corresponding bits in the *INTERRUPT_TARGET* register (0x80 sets the MSBit, and so forth), and if the corresponding *INTERRUPT_MASK* bit is set and the result is nonzero then the local module *INT** signal is activated.

The Texas Instruments chipset provides a more complete set of registers for activating local interrupts in response of general system events. These will be explained later. Among them there is a bit signalling the standard Futurebus+ *Power Fail* arbitration message [Futurebus+ 91].

Since the mechanism for setting interrupts is through writes to CSR space or through arbitration messages, it is clear that only those modules that are capable of arbitration will be able to activate interrupts. Therefore, any I/O device that needs to synchronise through interrupts must be arbitration capable.

5.3.3 Arbitration Messages

The DPU only allows distributed arbitration messages as defined in the [Futurebus+ 94a] and [Futurebus+ 91] standards. These messages are broadcast over the arbitration bus, and received by modules that are capable of arbitration concurrently with normal Futurebus+ data transfer activity. Messages that are not targeted interrupts are stored in a local FIFO if their message bit pattern meets the criteria defined by two sets of message mask registers. These two sets allow classification of messages into two classes. Whenever a message is received in the local FIFO, the local *INT** signal is activated. *INT** is also activated when a local interrupt is signalled, or when the message FIFO is full.

Messages 0xfe and 0xff are not maskable; the latter is defined in the [Futurebus+ 94a] standard as *PFAIL* (*Power Fail*). Messages can be prevented from being lost by either stalling the arbitration process or signalling an arbitration error when a message is being broadcast and any local FIFO is full. The Texas Instruments chipset is also able to ignoring incoming messages, while still storing them in the local FIFO.

5.4 The Demonstrator

In order to demonstrate the assumptions made in FASST architecture and in the processor modules, a simpler machine, composed of two DPUs and several standard boards, has been constructed. The implementation of Figure 5.4 has two DPU modules, a memory module, and an I/O module that acts as a bridge between the Futurebus+ and a VME bus that contains a hard disk controller. Table 5.2 gives a more detailed description of the non-DPU components.

As previously intimated, the memory map is software reconfigurable and also depends on the geographical address of the module. Table 5.1 shows the memory map of the Demonstrator. The bus-bridge is inserted in slot 3, the memory in slot 4 and the two DPUs in slots 1 and 5.

Futurebus+ Global Memory				
# Slot	Board	Starting Ad.	Ending Ad.	Description
Slot 1	DPU	20 0000	28 0000	Local Shared Memory
Slot 2				
Slot 3	Technobox	600 0000	600 3FFF	Two-port memory
Slot 4	Nanotek	1000 0000	1FFF FFFF	Global Memory
Slot 5	DPU	A0 0000	A8 0000	Local Shared Memory

Table 5.1: Futurebus+ global memory map for the Demonstrator

5.5 DPU Prototype

The DPU is a fail-stop processing module intended to serve as building block for use in fault-tolerant shared memory multiprocessors, using the Intel i486/DX2 processor. Its most unique characteristic is its ability to detect

MUPAC 512 series FB+	5 slots Futurebus+ rack with power
NMEM-1	16 MB of memory Compliant with A, B and F IEEE 896.2 standard Futurebus+ profiles 32 bits addressing Cache coherence protocols implemented Data bus width of 32 and 64 bits
FBV68LC040	Futurebus+ to VME bus bridge Motorola 68LC040 (25 MHz) processor 16 KB two-port RAM of 16 KB (Addressable from 68LC040, VME and Futurebus+) 512 KB SRAM (Addressable from VME and 68LC040) RS232 Port IEEE 1394 serial interface
ATX-630	Motorola MC68030 processor 2 MB two-port RAM SCSI Controller Floppy disk controller 2 serial ports 1 parallel port Ethernet interface Watch-dog 2 programmable timers

Table 5.2: Demonstrator non-DPU components

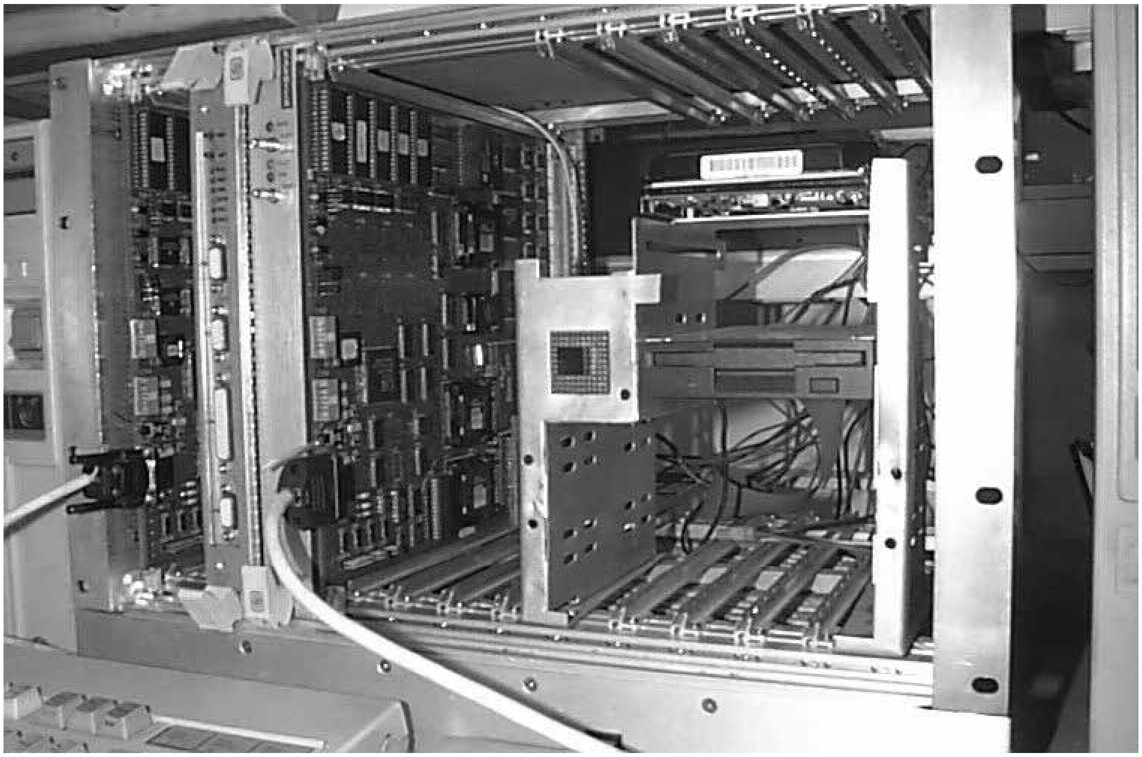


Figure 5.4: Two DPUs in the Demonstrator chassis (the DPUs are the modules with cables attached)

errors and isolate itself from the system bus once they occur [Fabregat et al 94]. It is also able to be inserted into or withdrawn from a live system, and it has two independent *Test Access Ports* conforming to the JTAG specification.

There are various different mechanisms for error detection. Data parity errors, timeout errors and processor disagreement are all able to bring the DPU to the faulty state. In addition, errors on accesses to local resources are also detected and treated by means of an interrupt.

Having chosen the Texas Instrument chipset to interface the DPU to the Futurebus+, and given the requirements imposed by the Futurebus+ standard, the DPU consists on a set of subsystems built around the several busses of the chipset (see Figure 5.5), joined together by means of several EPLDs (which are key components in the design).

Two of the subsystems relate to the i486/DX2 system interface: the *Dual Processors and Compare Logic*, and the *Cache Memory and Control*. Two others, the *Futurebus+ Interface* subsystem and the *Local Memory*, relate to the *Host Bus*, which is a local bus defined by the Texas Instrument chipset. Finally, there are global *Clock and Reset* and *Diagnostics* subsystems. The interface between the i486 system bus and the *Host Bus* is performed by the *FB_FPLA* EPLD.

The Texas Instrument chipset also defines a 8bit *CSR Bus*, where, in addition to the standard CSR registers and some non-standard ones required by itself, it expects any ROM or 8 bit I/O devices to be attached. The interface between the *Host Bus* and the *CSR Bus*, as well as part of the memory decoding task, is carried out by the *CSR_LMC_FPLA* EPLDs.

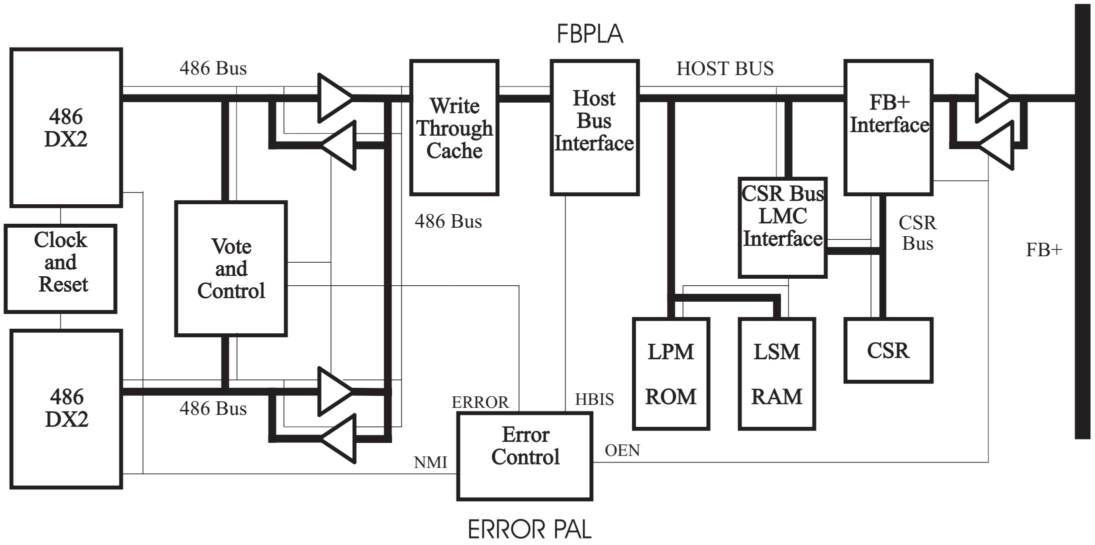


Figure 5.5: Block Diagram of the DPU

The core of the DPU is a pair of Intel i486/DX2 processors working in lock-step, whose busses are compared to provide the fail-stop characteristic to the system. Each set of i486 signals is driven independently to each of the comparators, and converted into a *Single i486 Bus* by means of some appropriate transceivers and buffers that allow incoming signals to reach both i486s, while selecting only one of them to provide outgoing signals to the system. One of the EPLDs that form the comparators is also in charge of activating the appropriate signals to control the transceivers. Beyond the transceivers, and connected to the *Single i486 Bus*, are two cache subsystems, one per i486. These are based in the HT44 cache controller and form a write-through, direct-mapped, read-allocate cache.

The rest of the DPU resources are located in the *Host Bus* part of the system, thus interfacing between both busses is required. The *FB_FPLA* performs this task, plus some others. The DPU module has 512KBytes of *Local Private Memory* and 128KBytes of *Local Shared Memory*, both connected directly to the *Host Bus*, and mapped to the lowest addresses of their respective spaces.

The *Host Bus* has only two possible masters; one is the *FB_FPLA* translating i486 requests; the other is the Texas instruments chipset, comprised of the TFB2002, TFB2010 and TFB2022 devices. The TFB2002 performs most of the *Host Bus* handshake protocol, while the TFB2022 supplies and receives data to/from both the *Host Bus* and the Futurebus+. The TFB2002 and the TFB2010 are also connected to the *CSR Bus*, as they contain some CSR registers. The TFB2022 is in charge of address decoding for both the *Host Bus* and the Futurebus+ transactions, and so it manages its CSR registers using the *Host Bus*.

The *CSR_LMC_FPLA* controls partial local memory decoding and accesses, and *CSR Bus* interfacing. It relies on the decoding information supplied by the TFB2022 to fully decode local memory and CSR space addresses. It provides the protocol to interface local memory and *CSR Bus* accesses with the *Host Bus* protocol.

All the local resources other than the local memory are mapped in the CSR space, and connected to the *CSR Bus* (apart from the TFB2022 CSR registers). There are a DS1397 Timer, an 82510 UART, a 74BCT8373 register, an 82C59-A Interrupt Controller, the *Capability ROM* (an 8Kx8 bits PROM), and the CSR registers within the TFB2002 and the TFB2010ABC.

The system of Figure 5.4 contains two DPU modules, a Nanotek NMEM-1 16MBytes memory module, and a Technobox FBVL68LC040 Futurebus+ to VME bridge, where some standard VME communication ports and SCSI Hard Disk controllers are connected. Everything is housed in a 5-slot MUPAC 512-series powered Futurebus+ rack.

5.5.1 Intel i486/DX2 Bus Subsystem

This subsystem has two active devices: the two i486 microprocessors, *i486A* and *i486B*, working in lock-step. Any disagreement between them must cause the module to stop and, in some cases, to isolate itself from the Futurebus+.

The Intel i486/DX2 has a 32bit synchronous system bus. The memory space is byte-addressed by 32 logical address bits that are translated into 30 address signals and 4 byte enable signals. Data and address busses are not multiplexed; they working in parallel during transfers, and can support overlapped address and data transfers. The bus allows for single, multiple and burst transfers, and the type can be dynamically negotiated between the processor and the slave during the transfer. The bus is mastered by the processor, and released in response to a request by an external device. It also supports locked and pseudo locked cycles, interrupt acknowledges and several other special cycles (cycle backoff and restart, cache invalidation...)

All data and parity signals are connected to the data bus without modification. The parity status pin is fed directly into the *ERROR PAL*, which controls the response to errors. Although the bus width can be dynamically adjusted, in the DPU it is always 32bits wide. 8bit devices are located on the *CSR Bus*, and accesses to them are converted to 32bit values as described later. The byte enable signals are used by the *FB_FPLA* to generate the low order address signals for the *Host Bus*. The *FB_FPLA* also uses the i486 cycle definition and control signals to generate the corresponding activity on the *Host Bus*, and interfacing to the cache subsystem.

5.5.2 Comparators and error control

This subsystem contains five EPLDs that perform the tasks of comparing the i486 busses to detect errors, of signalling any such errors to the *ERROR PAL*, and of managing the interface between the two i486 busses and the *Single i486 Bus*.

Four of these EPLDs behave as comparators. Data and address comparison has been duplicated, using complementary logic, to give four pairs of error outputs, each of them coded in a 1-of-2 code. *ERRa[0:1]* and *ERRb[0:1]* signal an address error, while *ERRa[2:3]* and *ERRb[2:3]* signal a data error. These comparators fully adhere to the standard i486 bus protocol.

The fifth EPLD, the *Comparators Controller*, checks the outputs of the comparators to signal an error in any of the following cases:

(a) Any of the comparator pairs indicate an error

(b) Both comparators of the same pair give different results

(c) Any of the four error inputs do not follow the 1 of 2 code

This EPLD also uses some of the i486 control signals to validate the results of the comparison; this allows the comparators to match the speed of the processors without introducing wait states.

A pair of watchdog timers increase the fault coverage of the module, on per processor. These timers are meant to detect infinite loops that will not be detected by the comparators if both i486 execute the same code. These watchdogs are constructed from two cascaded counters, and a retriggerable monostable multivibrator. An error is signalled to the *ERROR PAL* when the monostable output level goes low. To avoid this situation, the counters are clocked by the i486 *LOCK#* signal, and the monostable retriggered off one of the four outputs of the high order counter. The *LOCK#* signal has been selected to control the time out as it is easily detected, frequently activated in a multiprocessor system and not likely to appear in a loop.

5.5.3 Cache memory and controller

The DPU module contains 256KBytes of write-through direct-mapped cache, with 16 data bytes per line and 12 tag bits. Each line has an associated valid bit.

The cache system is controlled by a Headland HT44 chip. Basically this chip responds with data on read hits, updates its contents on write hits, and allocates lines transparently during read misses. The system is only able of cacheing 1GByte out of the 4GBytes i486 address space. To select the cacheable area, the *Memory Mapping PAL* acts upon the *KEN** input of the HT44. Typically the *Memory Mapping PAL* is programmed to only allow cacheability of the lower 1GByte of system memory, since this is where *Global Shared Memory* (*GSM*) resides.

The 256KBytes of cache memory are implemented using two sets of four MCM62486 32K x 9bit SRAMs. Address signal *A17* is used to select the set. Each set stores 8K cache lines, so 8K x 13bits of tag storage is needed per set, as well as 8K validity bits. This is achieved with four P4C164 8K x 8bit SRAMs (two per cache data set). The HT44 does not provide parity over the tags. Address signals *A18* to *A28* are used to form the tags. The 12th tag bit is supplied by the Memory Mapping PAL, using *A29*.

5.5.4 *Host Bus* subsystem

The *Host Bus* [Texas Instruments 94] (or *Host Interface* according to the latest version of the Texas Instruments Futurebus+ Interface Family Data Manual) is a 32bit synchronous bus used to interface the Texas Instruments Futurebus+ chipset with the module built around it. It can also be configured as a 64bit bus, but only for burst transactions). The *Host Bus* provides 32 or 36 address signals, and allows for single and burst transactions. The characteristics of the later (speed, data bus width and transfer length) are negotiated between the master and the slave prior to the start of a transaction.

Arbitration is performed via bus request and grant signals, and a unique bus idle (bus grant acknowledge) wire-ORed signal. There is no arbitration policy specified, as the arbiter should be designed according to the local module requirements.

The bus includes a lock signal to force Futurebus+ locked transactions, and the ability to group multiple *Host Bus* transactions into a single Futurebus+ transaction. There is also an interrupt signal and an *IGNORE** signal to prevent local transactions from propagating to the Futurebus+.

5.5.5 *FB_FPLA* EPLD

The *FB_FPLA* is the name given to one of the most important parts of the design. It is an EPM7064QC100-10 EPLD that provides many different functions, the most important of which is to act as the interface between the i486 bus and the *Host Bus*. Its remaining functions are to arbitrate for the *Host Bus*, to manage the interrupt protocol, and to manage cache line invalidation.

The HT44 invalidates cache lines during non-cacheable read cycles in which the *FLUSH** signal is asserted, regardless of the hit status. The *FB_FPLA* participates in an invalidation procedure that also requires the collaboration of special invalidation software, and it detects i486 interrupt acknowledge cycles from the cycle definition signals (the rest of the interrupt handling procedure is carried out by the *CSR_LMC_FPLA*). The *FB_FPLA* is also the arbiter of the *Host Bus*, requesting the i486 bus whenever a *Host Bus* mastership request is issued. There are only two possible masters of the *Host Bus*, the i486 and the Texas Instruments chipset. The latter will request use of the bus only when the module is acting as a Futurebus+ slave, which is not likely to happen frequently. The arbitration obeys a *Release On Request* policy.

One of the most important tasks of the *FB_FPLA* is to provide an interface between the *Single i486 Bus* and the *Host Bus*, taking into account the type of transaction in progress, the adaptation of some address signals and some special behaviour given certain particular transactions. It translates every i486 read or write cycle, except cacheable reads, into the corresponding non-burst *Host Bus* cycle.

The DPU Local Memory is formed of 512KBytes of *Local Private Memory* (*LPM*) for ROM, and 128KBytes of *Local Shared Memory* (*LSM*) for RAM. The two sets of memory are mapped to the lowest addresses of the *LPM* and *LSM* address spaces, respectively, as indicated in the DPU memory map. This local memory is physically connected to the *Host Bus*. The TFB2022 decodes the memory space by means of the *MS[0:1]* signals, and the *CSR_LMC_FPLA* generates the corresponding control signals. Buffers are used to drive the address and control signals to the memory chips.

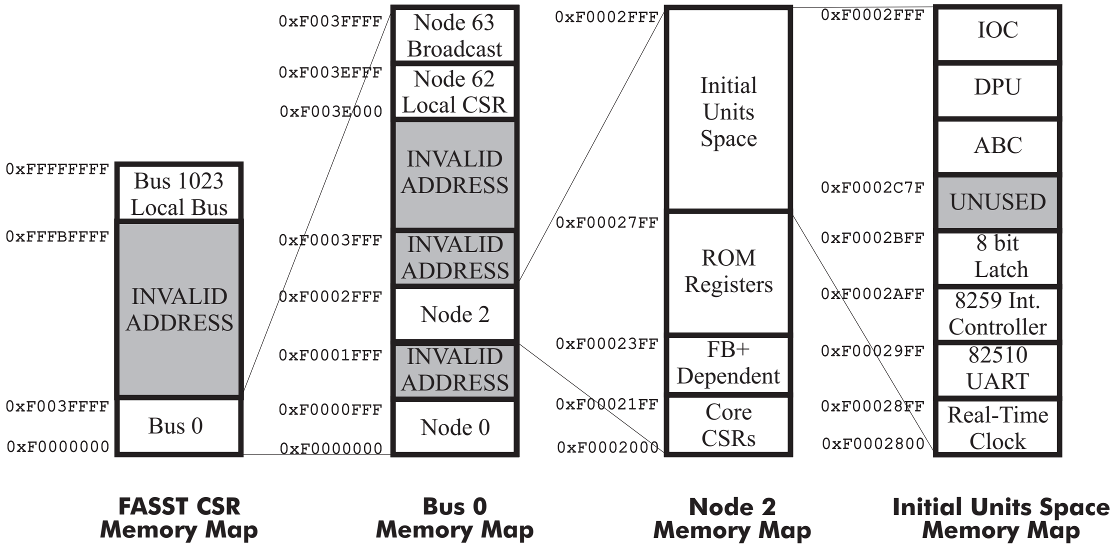
The ROM is composed of four AM29F010 128K x 8bit FLASH PROMs plus a fifth to store one parity bit per byte. To simplify the design, and given that the four parity bits are stored together, there is only one chip select signal for the five chips

The RAM is made of four CY7C188 32K x 9bit SRAMs, each one including a parity bit per byte. Each RAM chip has its own chip select and a common write enable signal. The *CSR_LCM_FPLA* also generates an output enable common to all the memory chips.

5.5.6 *CSR_LMC_FPLA* and *CSR Bus* subsystem

The Configuration and Status Registers (CSR) are a set of standard registers specified by the Futurebus+ standard. The standard specifies the CSR as a 32bit word-aligned address space (the two least significant address bits must be zero), that supports only 32bit addresses. Unimplemented locations must return zeroes when read, and incorrect writes (to read-only or unimplemented addresses) must be ignored. CSR registers should be accessed as 32bit data transfers. The CSR memory map appears in Figure 5.6. As mentioned above, the Texas Instruments chipset specifies a simple 8bit bus to access these standard registers.

The *CSR_LMC_FPLA* controls the access to every local device connected to the *Host Bus* other than the TFB2022 CSR registers. It is divided into two main functional units, the *Local Memory Controller* (*LMC*) that controls local memory accesses, and the CSR controller (*CSR*) that handles accesses to CSR, decodes I/O port addresses, and interfaces with every device in the CSR address space (apart from the TFB2022 internal CSR registers). It is composed of an EPM7192QC160-12 and an EPM7032LC44-12 EPLD.



FASST CSR Memory Map **Bus 0 Memory Map** **Node 2 Memory Map** **Initial Units Space Memory Map**

Figure 5.6: Bus 0, Node 2, CSR memory map

To perform the decoding tasks, the *CSR_LMC_FPLA* relies on the *Memory Space* signals provided by the TFB2022; these provide easier decoding and allow for programmable system global addressing. The *Memory Space* signals *MS[0:1]* distinguish between four different address spaces:

Local CSR Space This is the standard CSR address space.

Local Extended Units A programmable address space that in the case of the DPU module is used for the *Local Private Memory*.

Local Memory A programmable address space that in the case of the DPU module is used for the *Local Shared Memory*.

Futurebus+ Addresses The decoded address does not belong to any local resource, i.e. does not correspond to any of the three previous spaces. This combination of the *MS[0:1]* signals is a don't care transaction indication to the system. The TFB2022 signals this combination, rather than the *Local CSR Space*, during any transaction to its own internal CSR registers.

The CSR registers are not programmed after reset, and thus the decoding cannot rely on the provided signals at this stage. To accomodate this, the *CSR_LMC_FPLA* includes a simple *RESET* procedure that allows ROM reads and CSR programming before the Texas Instruments chipset becomes fully functional.

As each RAM memory chip stores its own parity bit, partial transactions are straightforward, activating only the addressed chip enables. If the access is to the ROM, all five (four plus parity) modules are selected, since all the parity bits are stored together, bearing in mind that the i486 to read only the bytes selected by BE[0:3]#.

The *CSR Bus* controller part of the EPLD is in charge of decoding accesses to the CSR address space, by selecting the appropriate device. Also it transforms the 32bit *Host Bus* data to the 8bit *CSR Bus* data and vice-versa, generating the corresponding address signals *CA0* and *CA1*, plus any parity bits not supplied by the devices (actually only the Texas Instruments device's CSR access supplies the parity bit, so the remainder must be generated by the EPLD). When accessing to CSR registers or the *Capabilities ROM*, the *CSR_LMC_FPLA* performs four *CSR Bus* accesses to consecutive locations. For simplicity, the rest of the CSR devices each have a byte address mapped onto a 32bit word boundary, and the *CSR_LMC_FPLA* performs a single access, supplying zeroes for the upper bytes during reads.

The *CSR_LMC_FPLA* detects several kinds of errors during accesses. These include read access to a non-implemented device or memory location, to a write-only device, or non-aligned CSR access, in which case the EPLD will supply zeroes to the *Host Bus*; write access to a non-implemented device or memory location, to ROM or a non-aligned CSR address, in which case the write will be ignored; read or write accesses to *Local Memory* where the size of the transaction and the least significant address signals form a non-valid combination, in which case an interrupt is signalled via the *IR5* signal of the *Interrupt Controller*. The EPLD also controls timing for every device access, as well as generating the required handshake signals for the *Host Bus*.

As the I82C59A resides in the *CSR Bus*, the *CSR_LMC_FPLA* collaborates with the *FB_FPLA* during interrupt acknowledge cycles to allow the 8bit interrupt number to reach the corresponding i486 data byte in keeping with the i486 interrupt acknowledge protocol.

In addition to the TFB2002 and TFB2010 standard and non-standard CSR registers, there are a number of other devices located in the CSR address space and physically connected to the *CSR Bus*. A brief description of these devices and their applications in the system follows.

Capability ROM This is a 1K x 8bit ROM that holds the standard values specified in the IEEE P1212 documents [CSR 91a, CSR 91b]. Most of these values hold the defaults to be loaded in the CSR registers to allow proper system operation. Some module identification information is also stored in this ROM.

DS1397 Real-Time Clock The DS1397 is a real-time clock device that contains 64bytes of data, including registers and some uncommitted RAM. It generates an *IR4* interrupt signal from three different sources: an alarm interrupt, a periodic interrupt, and a cycle-end interrupt.

82510 UART The 82510 is a well known dual serial port controller. It activates the *IR6* interrupt signal when any of interrupt conditions occur.

74BCT8373 Latch This device is a write-only register used to store various system information and configuration bits. Bits 0 to 2 are used to control the illumination of the Futurebus+ *SWAP*, *FAULT* and *RUN* front panel LEDs. Bit 5 feeds the signal *PATHSEL* used to select the i486 that is to propagate its outputs to the *Single i486 Bus*; this bit is fed into the *Comparators Controller* as explained above. Bits 6 and 7 are the configuration signals *ERRSEL1* and *ERRSEL2* that modify the behaviour of the *ERROR PAL*, as also explained above. Bits 3 and 4 are not used.

I82C59A Interrupt Controller The I82C59A Priority Interrupt Controller handles all the interrupt activity of the DPU (with the exception of *NMI*, as explained above). It generates the maskable interrupt signal *INTR* to the i486, and supplies an 8bit interrupt vector for the interrupt acknowledge cycle. The interrupt acknowledge protocol is carried out with the help of the *FB_FPLA* and *CSR_LMC_FPLA*, again as described above.

Various devices are able to generate interrupts. These are outlined below:

IR0 TFB2010 *PFAIL**. The Arbitration Controller activates this signal upon reception of a *Power Fail* arbitration message.

IR1 TFB2010 *INT**. The Arbitration Controller activates this signal whenever any of the standard Futurebus+ interrupt mechanisms (*Targeted Interrupts*, *Arbitration Messages*,...) sets an interrupt condition in the module. Some internal TFB2010 conditions can also cause this signal to be activated.

IR2 TFB2002 *INT**. The I/O Controller activates this signal whenever any of the interrupt conditions as specified by the *ERROR HI*, *ERROR LO* and *INTERRUPT MASK ENABLE* CSR registers is active.

IR3 External *SWITCH*. This interrupt is activated when a user toggles an external switch.

IR4 DS1397 TIMER *IRQ*. The TIMER activates this signal.

IR5 *LMC_ERR*. The *CSR_LMC_FPLA* activates this signal to indicate a severe error as explained above.

IR6 82510 UART *INI*. The UART activates this signal whenever any of the internal interrupt conditions occur.

IR7 Not used.

5.5.7 Futurebus+ interface

The DPU module interfaces to the rest of the system, based on the standard Futurebus+, by means of the Texas Instruments chipset. This chipset conforms to the Futurebus+ profile B with the exception of a central arbitration. It is composed of three devices that perform the bus arbitration, protocol handshake and transaction management, plus a number of BTL transceivers.

The chipset behaviour is configurable using the required standard CSR registers and some other CSR registers specific to the chipset itself. The CSR registers are distributed (and in a few cases shared) among the three devices. A brief description of the devices, their functionality and their relation to the DPU module is given below:

TFB2002 Futurebus+ I/O Controller The TFB2002 performs protocol handshaking to both the Futurebus+ and the *Host Bus*. It translates incoming and outgoing transactions between both busses. It is also in charge of participating as requester in the *Host Bus* arbitration protocol. Several kinds of errors produced in the transactions are detected by the TFB2002 and signalled (if not masked) to the host system by the activation of the *INT** signal (connected to the Interrupt Controller *IR2* request input). This chip also generates the global system reset signal *SYSRESET** to indicate that a global system reset is required. It interacts with the TFB2010 and the TFB2022 to handle all incoming and outgoing transactions, and to the *CSR Bus* to allow access to its internal CSR registers.

TFB2010 Futurebus+ Arbitration Bus Controller The TFB2010 is connected to the Futurebus+ arbitration bus. It performs the distributed arbitration protocol to gain tenure of the bus (in distributed mode) and to manage arbitration messages. It signals Futurebus+ interrupts to the module activating the *INT** signal (connected to the Interrupt Controller *IR1* request input). It also activates the *PFAIL** signal (connected to *IR0*) on reception of a *Power Fail* message. The TFB2010 interacts with the TFB2002 to coordinate Futurebus+ requests and master signalling, and to the *CSR Bus* to allow access to its internal CSR registers.

TFB2022 Futurebus+ Data Path Unit The TFB2022 is connected to the Futurebus+ and *Host Bus* data and address busses. It performs address decoding and high speed data transfer tasks. It is provided with an internal FIFO memory to handle both compelled and packet modes, and the necessary CSR registers to perform address decoding according with the system memory map. It provides the address space selection signals *MS[0:1]* to the host module to simplify local decoding logic. Access to its internal CSR registers does not involve the *CSR Bus*, since that bus is managed by the TFB2022 itself. The TFB2022 interacts with the TFB2002 to handle all Futurebus+ and *Host Bus* data bus transactions.

BTL Transceivers Futurebus+ uses BTL standard logic levels in its signals. All the signals connected to the Futurebus+ must then be translated to BTL from the TTL levels used within the module. The BTL transceivers are also provided with special V_cc connections to allow, in conjunction with the Futurebus+ connector geometry, for live insertion and withdrawal. Three types of transceiver are used in the DPU; one of them, the *Competition Transceiver*, is a very specific devices that presents the module's competition number to the Futurebus+ arbitration bus.

The *ERROR PAL* generates the *ERROR1** signal that serves to isolate the module from the Futurebus+ by disabling the outputs of all the transceivers except the *Competition Tranceiver*, which is disabled by means of the *ERROR PAL*'s *OEB* output. This is specifically because this is a fail-stop processor module; after module failure the interface remains fully functional but is not able to compete for access to the Futurebus+ and thus is precluded from propagating its erroneous behaviour to the system.

5.5.8 Support and Miscellaneous Logic

A MAX700 chip is used to generate two system reset signals, one active-low and one active-high. Each of these is fed into a buffer and generates four pairs of buffered reset signals, *MRESET[1:4]* and *MRESET[1:4]**. The MAX700 is used to ensure the minimum reset period is met. It also asserts *RESET* as soon the power supply voltage decreases below an acceptable limit. A *RESET* can also be triggered from the *SYSRESET** signal generated by the TFB2002, or by the boundary scan *TRST* signal.

The DPU requires two primary clock frequencies. These are 33MHz for the i486/DX2 processors and *Host Bus*, and 40MHz for the Texas Instruments chipset as a clock for its Futurebus+ interface. These frequencies are generated from a single AV9194 oscillator. In addition, these frequencies can be changed by means of a DIL switch, principally for debugging. To reduce clock skew, and also because there are two processors and thus two different areas to propagate the clock signal to, the 33MHz clock is divided by 2 or 4 (selected by another DIL switch, but normally 2) before being propagated to the processors. The divided signal will have half of the skew and can be propagated with less problems than the original one. At the processors, CYB992 clock drivers multiply the clock back to full frequency. These drivers are located nearest to the critical components. To solve possible skew problems due to trace length, small additional skew variations can be introduced using some other DIL switches.

To assist in debugging and testing, and also to test their usefulness, five 74ACT8994 *Digital Bus Monitors* have been connected in parallel with the *Host Bus*. These devices can be programmed to trigger on certain events and store the result internally to a limited depth. They are set up and interrogated via a boundary scan loop, but because of the complexity of the setup and the possible volume of data being returned, a separate boundary scan loop is used to service them. Logically the two loops would be in operation at different times. The main boundary scan loop is intended to be used early to test that data paths are correctly working. Having proven the electrical operation of the card, the DPU software can then be debugged, using the *Digital Bus Monitor* to monitor certain transactions to see that they complete as expected.

5.5.9 Error detection levels

As was indicated above, in order to obtain high dependability in the system, the data processing modules have to detect errors as soon as possible to prevent corrupted data from contaminating the non-faulty parts of the system, i.e. they must exhibit the fail-stop behaviour previously described. The design of the fault-tolerant mechanisms and algorithms of the DPU must be very carefully done to ensure both low fault detection latencies and a high fault detection coverage. Let us now examine the different error detection levels that have been included in the prototype DPU. Figure 5.7 shows the structure of the hardware involved in this schema:

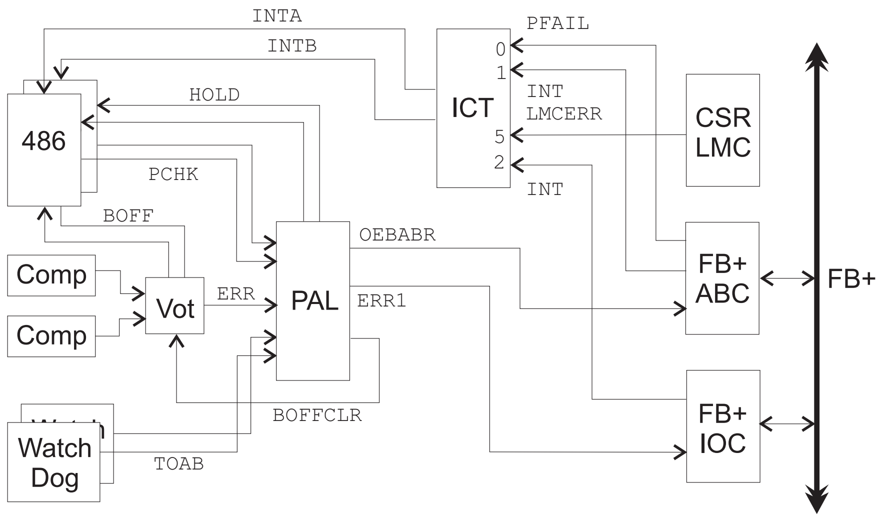


Figure 5.7: Error detection hierarchy

(a) In the first level, the processor performs its basic fault detection. Whenever it raises an exception due to an error in the execution of the current instruction, the processor is halted after showing diagnostic information on the system console.

(b) The module is a fail-stop one, incorporating two Intel 486/DX2-66 processors running in lock-step. Both processors execute the same instructions with identical data and most signals are compared at every clock cycle in order to detect errors. In the event of an error it is assumed to be a transient condition and the processor activity is stopped by asserting the signal *BACKOFF#* for 16 cycles. Following this the last cycle is restarted to recover the transient fault. If the error is reproduced again then the Futurebus+ output transceivers and the arbitration transceivers are disconnected from the system bus.

(c) In order to detect comparator errors, inverted logic is used to build pairs of comparators that perform the same function. In consequence, the main voter receives the information regarding processor errors in a duplicate complementary code that allows differentiation between three possibilities: *OK*, *error in the processors* and *error in the comparators*. If an error occurs the module is disconnected from the system and the processor held inactive.

(d) Both data buses are protected with parity. The i486 checks these bits at the end of every read cycle and asserts the signal *PCHK#* in the case of an error, which isolates the module from the system.

(e) Errors that affect the normal instruction flow of the processor are detected by the timeout of one or both of a pair of watch-dog timers. The watch-dog internal counters are reset by assertion of the processor *LOCK* signal. The operating system must be programmed to periodically activate this signal once the watch-dogs are enabled. Again, an error results in the isolation of the module.

(f) A memory controller monitors the accesses inside the address space of the module, and interrupts the processor when an invalid address or an invalid transaction is detected, thereby aborting the transaction.

(g) The last protection level is part of the Futurebus+ chipset. These circuits incorporate a complex protection and monitoring facility that may be programmed via the CSR registers to interrupt the processor in the case of an error. The processor can poll the state of the CSR registers within the interrupt routine to determine the origin of the error.

5.6 DPU Demonstrator software[2]

The DPU incorporates a software monitor in EPROM in order to:

(a) Allow access to the whole addressing space of Futurebus+, including I/O and CSR registers, in order to configure the hardware of the system (interrupts, alarms, RTC, serial port, ROM, RAM and Futurebus+)

(b) Implement a set of low-level procedures within ROM to ease the application programming and driver implementation. In addition the amount of memory required by the applications is drastically reduced by using this ROM library.

On system restart, boot software in the EPROM switches the processor to 32bit mode, resets the segment registers and programs the interrupt vectors, performs a local memory test and finally, if everything is correct after the *Power-ON Self-Test* (*POST*), it displays a welcoming message. Following this it checks for any key pressed to start the monitor; otherwise the user's application is executed. The most important utilities provided by the monitor are:

(a) memory write, read, fill, compare and copy,

(b) write, read or dump the CSR registers,

(c) serial port configuration and checking,

(d) real time clock programming (time, alarms and interrupts),

(e) program loading and debugging (in S-Record format), and

[2] The following section contributed by Carlos Pérez and Vicente Cerverón, Dpto. de Informática y Electrónica, Universitat de Valencia, Av. Doctor Moliner, 50, 46100 Burjassot, Valencia, España,

(f) interrupt programming (enable register handlers and exceptions).

Another important alternative is the possibility of debugging the module using the GNU symbolic debugger (gdb), which allows remote execution through serial port. Three debug functions, `putDebugChar()`, `getDebugChar()` and `exceptionHandler()`, allow communication with the serial port and control of the interrupt handlers.

Figure 5.8 is a photograph of the Demonstrator, showing the two DPUs working in parallel, each executing its own user tasks. When one module detects an error, the system is reconfigured, isolating the faulty module and allowing the task to migrate to the other non-faulty module. This process is based on the establishment of recovery points, the sending of *I'm alive* messages and the use of watch-dog timers. The user can trigger an artificial failure by activating an external switch which forces the cancellation of message generation. A further activation of the switch returns the system to normal behaviour, simulating live insertion of the module.

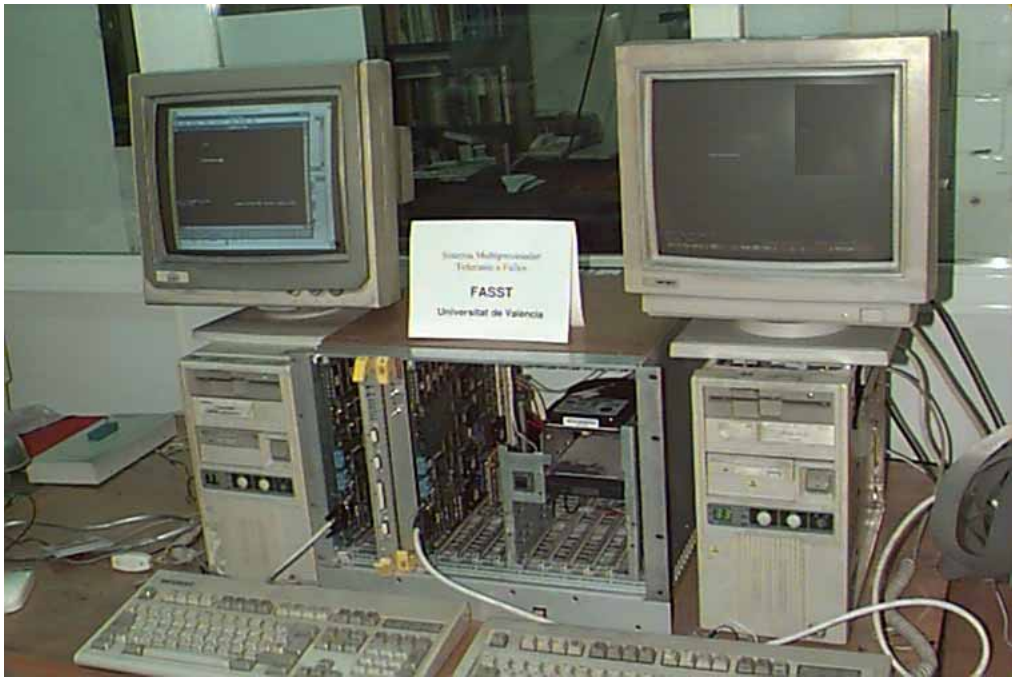


Figure 5.8: Photograph of the application execution environment

The software that perform the system recovery and reconfiguration has been split into two modules that are described in the two following sections.

5.6.1 Event-driven State Machine

An Event-driven State Machine (*ESM*) performs error detection and reconfiguration and builds an interface to communicate with upper levels. Hardware interrupts are converted to events (see Table 5.3), which produce changes of state. On power-ON one of the DPUs is elected as the *Monarch*, which is responsible for system reconfiguration and recovery when failure or live insertion occur. If the *Monarch* fails, the other DPU takes on that role. For each DPU, this state machine differentiates among the following states:

OK : The system is free of failures.

ALONE : The other DPU has failed and now I'm *Monarch*.

DEBUG : Test mode (I'm disabled during the test).

FAILURE : I've failed (until live insertion occurs).

READY : Debugging has ended or a new board has been inserted (I'm awaiting an *I'm alive* message).

When a module is live-inserted, it has to wait until the *Monarch* activates the master enable bit in the modules's CSR register *CSR LOGICAL_MODULE_CONTROL*. Following this the board goes to the *READY* state and begins to send *I'm alive* messages. The *Monarch* goes from the *ALONE* state to the *OK* state when it receives the first *I'm alive* message, and begins to send new messages. In the same way, when the new board receives the first *I'm alive* message it goes to the *OK* state. The changes of state can be seen in Figure 5.9.

System state is defined by four flags (*AmIAlive*, *AmIOnline*, *IsAlive* and *AmIMonarch*) and two variables (*Tics* and *TicsLastAlive*). *Tics* counts the elapsed time since system boot and *TicsLastAlive* holds the value of *Tics* when the last *I'm alive* message was received. The timer interrupt handler detects an error when *Tics* - *TicsLastAlive* is greater than a defined value, thereby forcing the state machine into the *ALONE* state.

Event	**Cause**	**Action**
I'm alive	*I'm alive* received	*TicsLastAlive* is updated
Alignment	*Monarch* detected live insertion	the new DPU is configured
Timeout	error detected	system recovery is initiated
Exception	debugger exception	debugger is started
End of debug	debugger ended	demonstration is continued
Power failure	power failure	DPU is isolated from the system
nmi	error detected	DPU is isolated from the system
Live insertion	live insertion	DPU waits for *I'm alive* message
Start-up	power-up	DPU waits for response after having sent *I'm alive* messages

Table 5.3: *ESM* Events

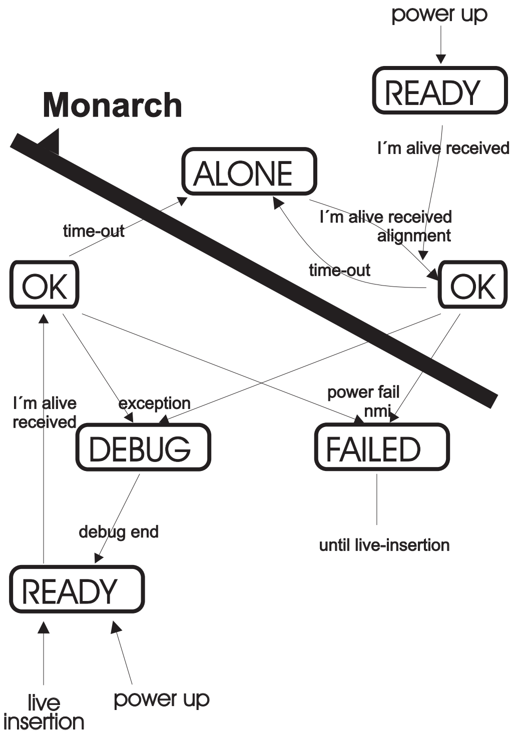


Figure 5.9: *ESM* module structure

5.6.2 Application Executor

The Application Executor (*AE*) module does not deal with events nor system messages. It uses the state of the *ESM* to drive the user application, and can handle system reconfiguration through task migration since it establishes recovery points in shared memory. Depending on the state of the *ESM*, the *AE* performs the following tasks:

OK : The *AE* executes the DPU user application.

ALONE : The *AE* recovers the state of the task of the failed module from the last recovery point and shares the processor time among the two tasks.

ALONE → **OK** : The *AE* informs the *ESM* that a new module is ready to continue with the task from the last checkpoint.

The module first checks for the detection of an alignment, indicating that the module has been live inserted. If this is not the case, then a system start-up must be in progress, so the monarch election phase begins. Every module sends an arbitration message with its geographical address and waits for the arrival of a new message which indicates the new *Monarch*. The new *Monarch* performs system configuration, programs the memory map and initializes the CSR's of the other boards, allowing Futurebus+ accesses. If a module has been live inserted, it has to wait until the *Monarch* sets the corresponding bit of the CSR register to enable bus mastership.

5.7 Evaluation of the dependability of the DPU Demonstrator[3]

One of the most important aspects to consider in the design of a general purpose fault-tolerant computer is the evaluation of the dependability of the system [Laprie 90]. The main objective must be to demonstrate that the behaviour of the system conforms to its specifications. This process is an essential prerequisite to knowing how well the system will tolerate faults; it also quantifies some parameters that allow comparisons to be made with other systems. Any evaluation of fault-tolerant characteristics within the FASST architecture must recognize that the dependability of the system strongly depends on the existence of fail-stop processors, and also depends on the rollback recovery capabilities of the remaining processors. When a DPU fails, firstly it has to be isolated from the system in order to prevent it from corrupting the remainder of the system, and then eventually the other modules have to detect its failure in order to begin the reconfiguration and recovery of the system. In this way, the evaluation strategy needs to measure two important sets of parameters:

(a) Error detection latencies and coverages

(b) System reconfiguration latencies and coverages

These parameters are required to distinguish between several dependability models of the system, so that we may calculate some dependability attributes, such as reliability, safety and availability.

We can apply many techniques to evaluate the system, depending on the level of the models and the set of parameters and functions we want to calculate. [Siewiorek et al 92] considers these aspects in depth. In the following we choose to use Markov models to estimate reliability, availability and safety, since they easily allow the inclusion of the coverage parameters [Carter et al 71] and also take into account the repairing process.

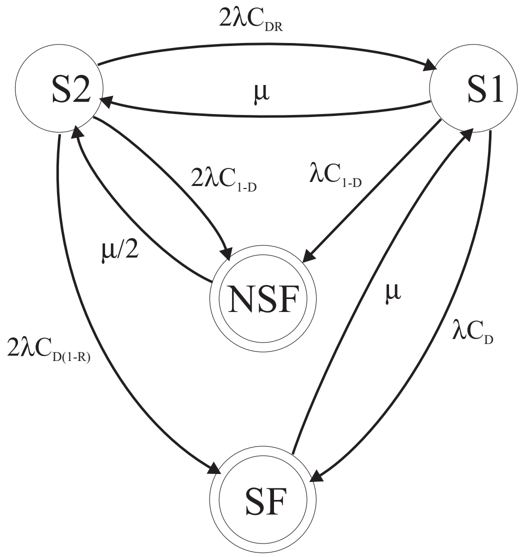
A Markov chain is composed from states which represent the system description (in this case the state will show the number of processor modules that have failed) and a set of transitions based on probabilities which drive the changes of state. Figure 5.10 depicts a macroscopic model of a fault tolerant system with two modules that allows functional degradation and repair; it also takes into account any fail-stop violations of a DPU. The model represents the essence of the DPU Demonstrator.

In state *S2* the two DPUs are working perfectly. Let us assume that eventually a failure occurs. If the error is detected then the model evolves to state *S1*, where the system is also operative but degraded. If the error is not detected, we suppose that the system fails in a non-safe way, since we cannot predict the consequences of the error, and the model evolves to state *NSF*. Lastly, the model evolves to state *SF* when the system safely detects the error but cannot reconfigure itself, and as a consequence is inoperative.

In order to estimate the reliability function of the system, let us assume that when a DPU fails (the model enters state *S1*), it can be live-withdrawn, repaired and live-inserted back into the system in 1/μ hours. An analytical solution of the chain will then give the probability of being in states *S1* or *S2* as a function of the failure and repair rates λ and μ:

$$R(t) = p_{S2}(t) + p_{S1}(t)$$

[3] The following section contributed by Rafael J. Martínez Durá, Instituto de Robótica, Universitat de València, Hugo de Moncada 4, Entlo., 46010, Valencia, España, and Pedro Gil, Departamento de Ingeniería de Sistemas, Computación y Automática, Universidad Politécnica de Valencia, Camino de Vera, s/n, Valencia, España.



Parameters	Meaning	Value
C$_D$	Fault detection coverage of the DPU	69.7 %
C$_{1-D}$	Fault no-detection coverage of the DPU	8.3 %
C$_R$	Error recovery coverage of the system	72.7 %
C$_{DR}$	Safe error recovery coverage of the system	67.8 %
C$_{D(1-R)}$	Safe error no-recovery coverage of the system	1.9 %
λ	Failure rate	
μ	Repair rate	

Figure 5.10: Macroscopic model

$$= \frac{k_1 t\ e^{-k_2 f_1(\lambda,\mu) - k_3\lambda t - k_4\mu t}(k_5 f_1(\lambda,\mu) - k_6\lambda - k_7\mu)((k_8\lambda + k_9\mu)f_1(\lambda,\mu) - f_1^2(\lambda,\mu))}{(k_{10}\lambda + k_{11}\mu)f_1^2(\lambda,\mu)}$$
$$- \frac{k_1 t\ e^{+k_2 f_1(\lambda,\mu) - k_3\lambda t - k_4\mu t}(k_5 f_1(\lambda,\mu) + k_6\lambda + k_7\mu)((k_8\lambda + k_9\mu)f_1(\lambda,\mu) + f_1^2(\lambda,\mu))}{(k_{10}\lambda + k_{11}\mu)f_1^2(\lambda,\mu)} \quad (5.1)$$

where any k is a constant, and:

$$f_1(\lambda,\mu) = \sqrt{k_A\lambda^2 + k_B\lambda\mu + k_C\mu^2} \quad (5.2)$$

The availability of the system can be calculated in the same way, but with care to avoid the absorbing states of the model adding system repairing rates from the states *SF* and *NSF* to the states *S1* and *S2*. A similar procedure can be applied to obtain the safety of the system but by considering the probability of being in the states *S1*, *S2* and *SF*. The computations can be simplified by using the SHARPE software [Sahner et al 95]. In the steady state the equations for availability and safety reduce to:

$$Availability = p_{S2} + p_{S1} = \frac{k_{12}\lambda\mu + k_{13}\mu^2}{f_2^2(\lambda,\mu)} \quad (5.3)$$

$$Safety = p_{S2} + p_{S1} + p_{SF} = 1 - p_{NSF} = 1 - \frac{k_{14}\lambda^2 + k_{15}\lambda\mu}{f_2^2(\lambda,\mu)} \quad (5.4)$$

where:

$$f_2(\lambda,\mu) = \sqrt{k_D\lambda^2 + k_E\lambda\mu + k_F\mu^2} \quad (5.5)$$

In order to quantify these functions, it is necessary to estimate the values of λ, μ, C$_D$, C$_{1-D}$, C$_R$, C$_{DR}$ and C$_{D(1-R)}$ by means of a process called experimental validation [Iyer 95]. This can be done at several points in the life of a system: in the design phase a system can be simulated to obtain the parameters; when a prototype is available it can be stimulated with synthetic faults, using a complex process called fault injection [Hsueh et al 97], and the consequences measured[4].

The DPU Demonstrator model has been validated using a pin level fault injector designed at the Universidad Politécnica de Valencia [Gil et al 97], plus a DAS (Digital Analysis System) to avoid the noise induced by the high frequencies and to trigger several events in the same injection. Figure 5.11 shows a diagram of the injection environment.

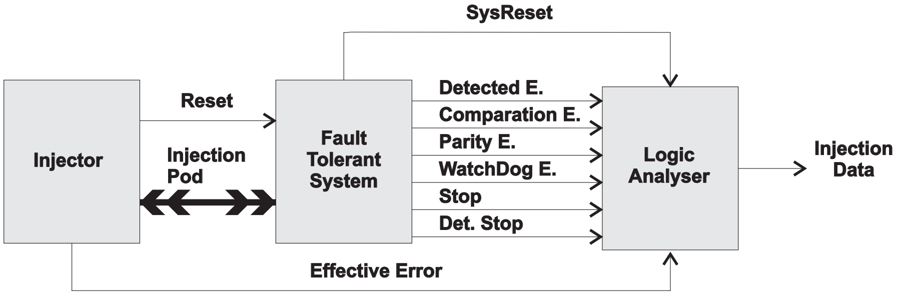


Figure 5.11: The injection environment

The main purpose of validation experiments is to calculate the fault detection and recovery coverages, and to measure the latency time of the fault tolerant algorithms and mechanisms within the system. In the following paragraphs we show the progression towards the calculation of the most important dependability parameters of the system. The experiment involves injecting a number of stuck-at faults in the master *i486A*, the slave *i486B*, the Host Bus and the CSR Bus. The faults can be classified as permanent, transient or intermittent, according to the time of activation of the fault. Table 5.4 shows the experimental results.

		Injection point			
		Master	Slave	Host Bus	CSR Bus
Kind of Fault	Permanent	2640	2640	750	630
	Transient	2640	2640	750	630
	Intermittent	2640	2640	750	630
	Total:	7920	7920	2250	1890
				Total injected faults:	19980

Table 5.4: Injected faults

Once the fault is injected, it is possible to determine whether the fault is detected by the comparators, by parity or by the watch-dog timers included in the DPU, and also to measure the latency of the comparators in the detection of the error, the time it takes to halt the DPU (in order to analyze the fail-stop characteristics), and the latency in the reconfiguration of the system. The results show that the system detects the fault 72.1% of the time and successfully reconfigures 75.1% of the time. Figure 5.12 shows the coverages.

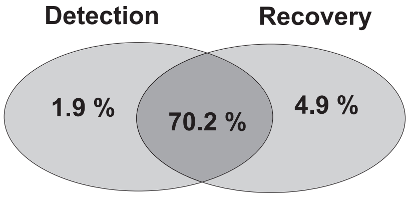


Figure 5.12: DPU Demonstrator coverages

The next step is to solve the analytical model discussed above, which depends on the failure and repair rates, λ and μ, of the modules of the system. Although these parameters can be roughly estimated using several reliability standards (such as MIL-HDBK 217F), it does not make much sense to solve the model to obtain single numbers in

[4] The table of Figure 5.10 shows the values of the coverages obtained from fault injection experiments.

order to show the dependability of the system; it is more useful to compare several systems for similar failure and repair rates, solving the respective models with the same values.

One of the most important steps in the experimental validation is the measurement of the latency times of the error detection, module halt and system reconfiguration. This yields direct observation of the fail-stop characteristics of the module, and also the real time capabilities of a simple application. Table 5.5 shows the measures acquired in injection experiments:

	Comparator	*Parity*	*Watch-Dog*	*Global*
Error Detection	528,0000	5242,000	1145,000	810,0000
Module halt	6697,500	5242,000	1145,000	5170,000
System recovery	6,27E+08			

Table 5.5: Median of the latency times in nanoseconds

To finish the analysis, the following graphs show the distribution functions of the several fault coverages versus time, and also the reconfiguration time of the system. These show the fault detection coverage of the comparators, parity errors, and also of the errors that modify the normal instruction flow (which are detected by the watchdogs). Figure 5.13 shows a comparison of the three detection error mechanisms. The abcises show the elapsed time from the activation of a effective error to the detection of the error. The ordinates measure the percentage of cases where the error has been detected before the elapsed time.

The comparators exhibit the best coverage and the least latency time, as could be expected, since they have been designed especially for the DPU. Adding the three functions yields the asymptotic fault detection coverage of the DPU, covering about 70% of the injected faults.

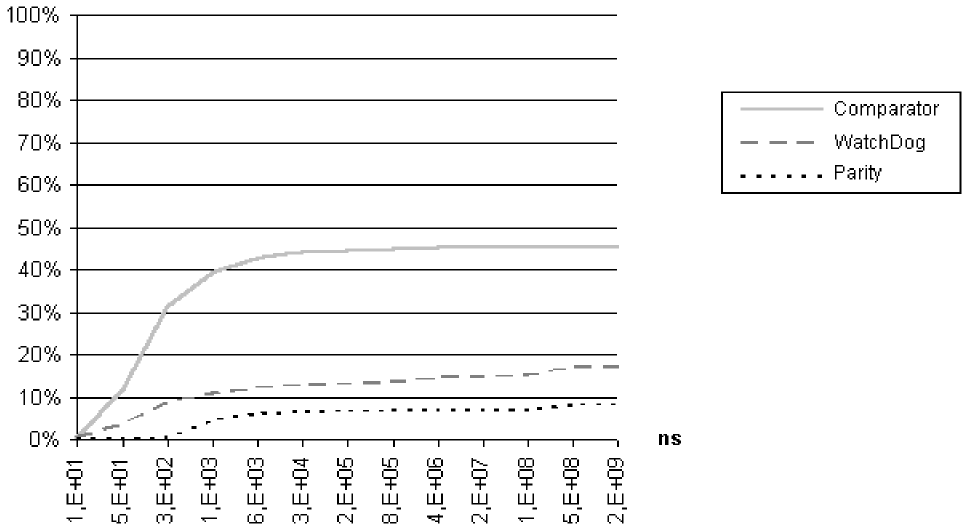


Figure 5.13: Error detection coverages versus time

The graph of Figure 5.14 shows the halt time of the DPU, i.e. the time elapsed from injection of the fault until the *HOLD* signal of the processor is activated, thereby isolating the DPU from the system. Again it is valuable to differentiate between the errors detected by the comparators, the parity protection mechanisms and the watch-dogs. The initial low coverage by the comparators is due to the activation of the transient error recovery mechanism when a comparison error is detected; the module performs a *BACKOFF* cycle for 16 clock cycles to avoid the system from halting due to a transient fault, and if the effect of the fault is still active when the processor executes the previous instruction the module is stopped - this causes the halt percentage to be very low for the first 300 ns after fault activation.

The last graph (Figure 5.15) measures the reconfiguration time of the system. This process begins with the error detection phase. When the timer that accounts for the arrival of *I'm alive* messages expires, the system reconfigures itself and the failed application is recovered. The figure depicts a histogram with the frequencies

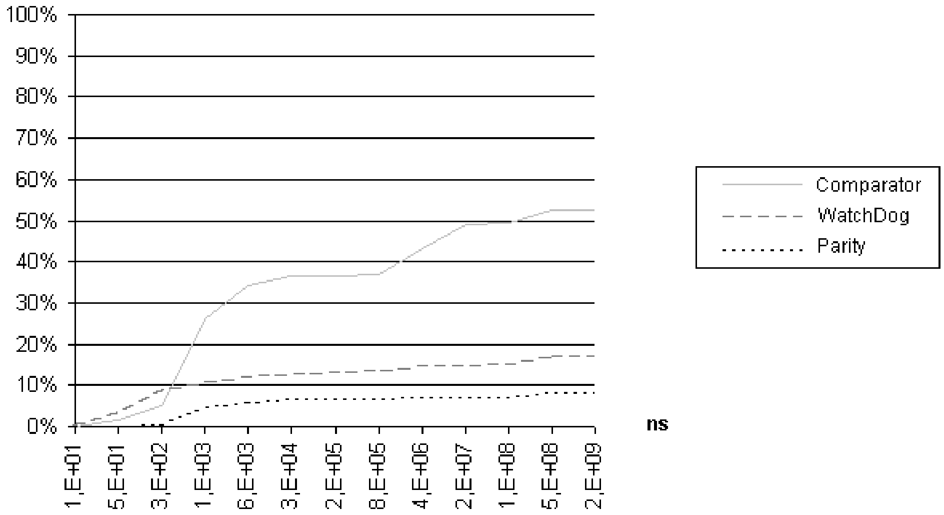


Figure 5.14: Module halt coverages versus time

of the several measured reconfiguration times, as well as the cumulative function of them, which represents the asymptotic recovery coverage of the system.

A more detailed description of the experiment can be found in [Martínez 97].

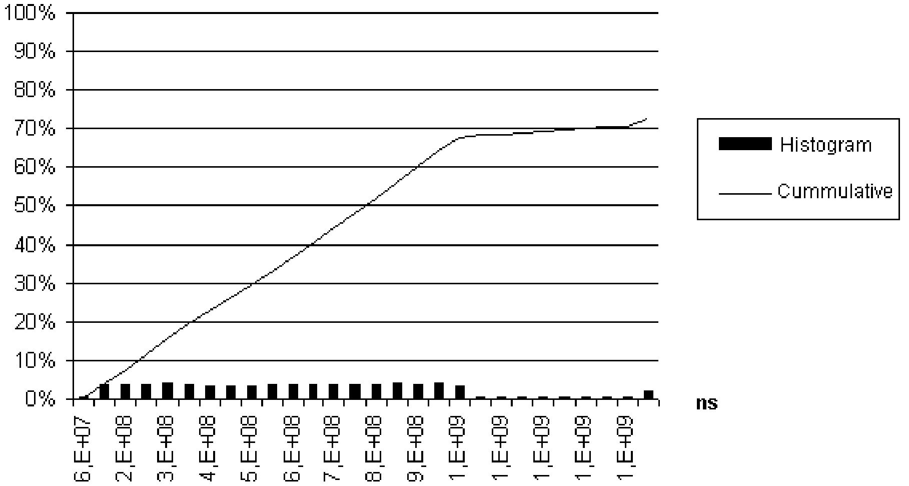


Figure 5.15: System reconfiguration coverages versus time

5.8 Summary

As can be seen, a fail-stop processor is not like a normal one; it requires considerable extra design effort and care, and a lot more logic. In order to obtain the dependability attributes of the system and to demonstrate the efficiency of the various algorithms and mechanisms, a complex high speed injection environment needs to be constructed so that physical fault injection can be used to test the behaviour of the system in the presence of faults. Futurebus+ does not contribute to this, but does provide a relatively protected environment within which errors can be detected.

Chapter 6

Secondary Storage

6.1 Secondary Storage[1]

Now we turn to an important secondary issue, not to stress the pun too far: that of secondary storage in a fault-tolerant system. The primary concern here is to extend the ability to tolerate processor failures to secondary storage, i.e. to extend the FASST architecture and recovery protocol to include secondary storage. This is done in three steps:

(a) By using redundancy in secondary storage.

(b) By designing the disk controller cache as stable memory.

(b) By incorporating the disk controller cache within the recovery protocol.

First let us look at the use of redundancy in secondary storage, now a standard way of introducing fault-tolerance, and a clear descendant of attempts to increase the performance of secondary storage. These attempts arose from the disparity between the rates of improvement of processor and secondary storage performances. Bill Joy predicted that processor performance would increase according to:

$$MIPS = 2^{year-1984} \tag{6.1}$$

While millions of instructions per second ($MIPS$) have long since fallen from favour, Joy's Law provides us with a useful rule of thumb; that processor performance approximately doubles each year. Advances in disk technology, relying on advances in mechanical and magnetic technologies, have shown relatively modest improvements as Table 6.1 shows.

Areal Density	27% p.a.
Linear Density	13% p.a.
Inter-Track Density	10% p.a.
Transfer Rate	10% p.a.
Seek Time	8% p.a.
Rotation Speed	8% p.a.

Table 6.1: Historical Rates of Improvement in Disk Technology

Amdahl's Law [Amdahl 67] calculates the effective speedup S of a system when some fraction f of the work can be performed in a faster mode, the speedup in the faster mode being k.

$$S = \frac{1}{(1-f) + \frac{f}{k}} \tag{6.2}$$

This implies that for a given application, if 10% of its work is I/O bound, then a 100-fold increase in processor speed, in the absence of any improvement in disk performance, will only increase the performance of the application by a factor of 10. In recent years, the growing disparity between CPU and disk speeds has been bridged by increased main memory size, caches on disk controllers and improved file system technology. All of these methods are based on the observation that disk access patterns demonstrate temporal and spatial locality; one of the most compelling application areas today, multimedia, often requires access to large amounts of data which are accessed in a sequential manner. Clever cacheing and buffering alone will not be able to provide the high, sustained data rates required by these applications.

6.1.1 Disk Arrays

One solution to the impending I/O crisis is to use multiple disks in parallel. This allows multiple small requests to be serviced simultaneously, while larger requests can be carried out more quickly by reading data from a number of disks in parallel. Cray Research used one such method, known as disk striping, to improve the I/O performance of their super-computers [Cray 85]. Disk striping involves interleaving data over a set of disks, that is, placing subsequent blocks of data on different disks. The size of a block of data may vary from one bit to multiple kilobytes.

[1] The following sections contributed by Brian Coghlan, Jeremy Jones, Danny Keogan and Philip O'Carroll, Department of Computer Science, Trinity College Dublin

For IBM, Kim [Kim 85, Kim et al 85, Kim 86, Kim et al 87, Kim et al 91] argued that these super-computing techniques could also benefit database applications. To address the concern for reliability in this context, she proposed extending the striping approach to include a checksum disk. This checksum disk would store error correction codes (ECC). She concluded that this approach would provide improved performance, parallelism through interleaving, and a uniform distribution of requests over a set of disks.

6.2 Redundant Arrays of Independent Disks (RAID)

In 1987/88, Patterson, Gibson and Katz, a group of researchers at the University of California at Berkeley, published first a technical report [Patterson et al 87], then a paper [Patterson et al 88] which built on Kim's work, providing an analysis of the reliability of an array of disks, coining the term RAID and defining a set of RAID levels. Disk manufacturers quote the reliability of their disks in terms of mean time to failure ($MTTF$). Patterson et al asserted that for an array of N disks, the mean time to failure for an array $MTTF_{array}$ could be calculated in terms of the $MTTF$ for a single disk $MTTF_{disk}$ thus:

$$MTTF_{array} = \frac{MTTF_{disk}}{N} \tag{6.3}$$

At that time, the $MTTF$ for a typical commodity disk was 30,000 hours, this meant that for a ten disk array the $MTTF$ would be less than 20 weeks. Clearly, redundancy in disk arrays was not an optional extra. Patterson et al. also developed an expression for the mean time to failure of a RAID, $MTTF_{RAID}$. If an array of D data disks is arranged into N_G groups each protected by C disks holding ECC information, and the mean time to repair of a disk is $MTTR$ then:

$$MTTF_{RAID} = \frac{MTTF_{disk}^2}{(D + G{\cdot}N_G){\cdot}(G + C - 1){\cdot}MTTR} \tag{6.4}$$

The above equation only deals with disk failures, and does not take into account failures in power supplies, cabling, controllers or software. A more detailed analysis of possible modes of failure can be found in [Schultze 88].

6.2.1 Taxonomy of RAID

Patterson et al defined a number of schemes governing the placement of data and ECC information over an array of disks. These schemas were referred to as RAID levels. Each RAID level attempts to redress some weakness or flaw in a previous one. Before discussing the details of individual levels, some general concepts and terminology will be introduced. The definitions below are slightly modified versions of those given in [E.K.Lee 90].

(a) A **sector** is the minimal unit of data that can be read or written to a disk.

(b) A **block** is the unit of data interleaving, the amount of data placed on one disk before placing data on another, also referred to as a stripe unit. A block is an integral number of sectors.

(c) A **stripe** is the set of blocks that ECCs are calculated over, also referred to as a parity group, or a parity stripe.

(d) A **rank** is the set of disks a stripe is placed on.

(e) A **string** is the set of disks formed by taking one disk from each rank; a disk may not belong to two strings.

Although it is common practise, particularly in software RAIDs, to attach more than one string to a disk controller, the string becomes a single point of failure, and so we will confine the discussion to configurations where every string is attached to a different disk controller.

Five RAID levels were defined. Additional levels have since been proposed, see [Katz et al 89, RAB 93].

6.2.1.1 RAID 0

RAID Level 0 could perhaps be more accurately referred to as AID 0 since this data placement scheme includes no redundant (ECC) information. RAID 0 improves performance through parallelism and incurs none of the performance or capacity overheads associated with redundant schemes. However, if one of the disks in the array fails, data will be lost. As stated in section 6.2 this gives a mean time to failure which is inversely proportional to the number of disks in the array. As the number of disks in the array increases, this quickly reduces the $MTTF$ of the array to an unacceptable level.

Data is striped over the drives. No parity is stored. The drive spindles may be synchronised, but this is not mandatory. Each successive block of the disk subsystem is stored on a successive drive, with wraparound. Thus logical block 1 will in all likelihood be on physical block 1 of drive 1, logical block 2 on physical block 1 of drive 2, logical block 3 on physical block 1 of drive 3, logical block 4 on physical block 1 of drive 4, and so on. At the time that each block is being accessed from its drive, the other drives are free for other accesses. This in no way precludes, say, a simultaneous access involving all drives. The advantage is that a number of disks can be seeking and transferring data in parallel, giving improved throughput. The disadvantage is that since there is no parity or any other form of redundant information stored, there is no tolerance to disk faults nor any increase in availability.

6.2.1.2 RAID 1

RAID Level 1 mirrors each disk in the array to provide redundancy and thus improve on the reliability of RAID 0. Data is still interleaved onto the disk in a similar way to RAID 0, but each disk now has a mirror disk. Data is written to both a disk and its mirror. Reads can be distributed to either disk so as to balance the load between the two. The advantages are that this gives the very best tolerance to disk faults and availability of all RAID levels, and that pairs of disks can be seeking and transferring data in parallel, giving improved throughput. The disadvantage is that this yields the most expensive level of RAID with the least storage capacity.

For RAID 1 disks are paired and the data is duplicated on each drive pair. No parity is stored, since 100% redundant data is already available. The drive spindles may be synchronised, but this is not mandatory. RAID 1 shows better read performance than RAID 0 due to the possibility of distibuting requests over more disks. Writes will incur the overhead inherent in transferring to two disks simultaneously. Recovery of a failed disk is accomplished by simply copying the contents of the remaining good disk onto a replacement disk. Data losses only occur in RAID 1 in the unlikely event of a disk and it's mirror disk failing simultaneously. RAID 1 effectively solves the reliability problems of RAID 0, but at an extremely high cost. Only half the storage capacity of the array is useable, doubling the cost per megabyte of storage space. The remaining levels cut down on this capacity overhead.

6.2.1.3 RAID 2

For RAID Level 2 the data is striped over the drives, with a number of redundant drives dedicated to storing a Hamming code [Hamming 50] formed by bit or byte-interleaved parity over the data on the other disks. The coding provides for single error correction and double error detection. This is similar to the ECC protection of data in memory. The drive spindles may be synchronised, but this is not mandatory.

The advantages are that the redundancy improves tolerance to disk faults and availability over RAID level 0, and that it is both less expensive and gives greater storage capacity than RAID level 1. The disadvantages are that other disks cannot be seeking nor transferring data in parallel (since all accesses involve the same redundant disks), and that every write involves a parity read-modify-write.

RAID 2 reduces the capacity overhead of RAID 1 by using the Hamming code. If data is bit or byte-interleaved onto a group of G disks, and C check disks, then for a given G the value of C is determined by choosing the lowest integral value for which the following inequality is true:

$$(G + C + 1) \leq 2^C \tag{6.5}$$

For example, for a group of 10 disks, 4 check disks would be required; a group of 25 would only require 5 check disks. Although RAID 2 reduces the capacity overhead of RAID 1 with minimal degradation of reliability, because it uses bit or byte interleaving, it requires that all disks in a rank be involved in each operation on that rank. This reduces the ability of the system to perform multiple requests in parallel, since only requests which involve different ranks can be executed simultaneously, causing small operations to be handled inefficiently.

6.2.1.4 RAID 3

RAID Level 3 further reduces the capacity overhead of RAID 2 by taking a slightly different approach to error detection and correction. Essentially, Hamming codes use one bit to indicate that a single bit error occurred; the remaining check bits identify which one. For a disk array, it is not necessary to encode all this information. If a disk fails, its controller will be aware of the failure, making the additional bits of the Hamming code superfluous. RAID 3 uses parity to detect single bit errors, the additional information provided by the controller allowing it to recreate the contents of a failed disk.

For RAID Level 3 the disks are accessed in parallel, with one redundant drive that stores the parity formed by exclusive-ORing the data on the other disks (see Figure 6.1). The drive spindles should be synchronised to allow all the disks to be accessed in parallel.

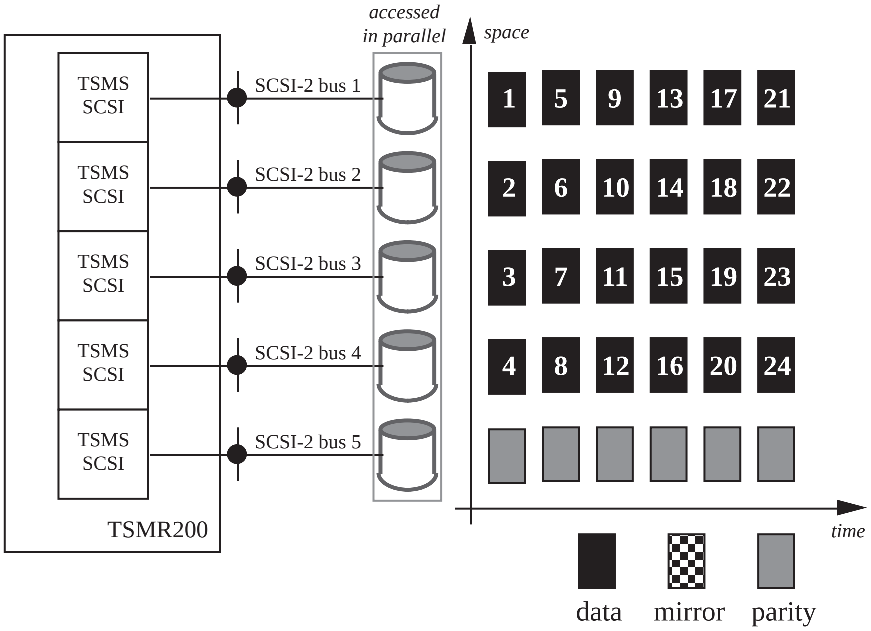


Figure 6.1: RAID Level 3

The advantages are that the redundancy improves tolerance to disk faults and availability over RAID level 0, that, along with levels 4 and 5, this yields the least expensive solution with the greatest storage capacity, and that parallel access to all the disks gives the highest block transfer rate of all RAID levels. The disadvantages are that there is no potential for benefit from other disks seeking in parallel (since all accesses involve all disks), that it gives poor performance with small block sizes, and that the spindles should be synchronised.

RAID 3 suffers the same performance impediments as RAID 2 since it also uses bit or byte interleaving. Both RAID levels perform particularly badly under workloads with many small operations, especially small writes. The advantage of RAID 3 lies in the fact that bit and byte interleaving systems are easy to implement in hardware, and applications which require streaming of data on and off disks in large blocks can be made to perform well.

6.2.1.5 RAID 4

Low performance for small operations under RAID 2 and 3 is purely an artifact of bit or byte interleaving. RAID Level 4 employs the block interleaving approach used by RAID 0 and 1 and combines it with parity for error correction. Data is striped over the drives, with one redundant drive that is dedicated to store the parity formed by exclusive-ORing the data on the other disks. The drive spindles may be synchronised, but this is not mandatory.

The advantages are that the redundancy improves tolerance to disk faults and availability over RAID level 0, and that along with levels 3 and 5, this is the least expensive solution with the greatest storage capacity. The disadvantages are that other disks cannot be seeking nor transferring data in parallel (since all accesses involve the same redundant disks), and that every write involves a parity read-modify-write.

RAID 4 is more reliable than RAID 0, has a lower capacity overhead than RAID 1 or 2, and performs better than RAID 3 for small operations. For large operations RAID 4 performs similarly to RAID 3. However, RAID 4 has one major disadvantage. All parity information for a rank is held on a single disk. This means that all writes to any portion of that rank of disks must access the parity disk. This disk can soon become a bottleneck.

6.2.1.6 RAID 5

In RAID Level 5, parity information is shuffled through the array of disks, as shown in Figure 6.2. Shuffling the parity avoids the problem of the parity disks becoming bottlenecks in the system. A variety of placement patterns have been proposed [Muntz et al 90, Merchant et al 92, Holland et al 92, Chen et al 95], and it has been suggested that for certain workloads parity placement may have performance consequences, both in failure-free state [E.K.Lee 90, E.K.Lee et al 91], and in failure state [Muntz et al 90, Chandy et al 93, Ng et al 92].

Data is striped over the drives. Parity is still stored on a drive, where the parity is formed by exclusive-ORing the data on the other drives, but no particular drive is dedicated to store the parity. Instead the parity associated with each data block is stored on a different drive. The drive spindles may be synchronised, but this is not mandatory.

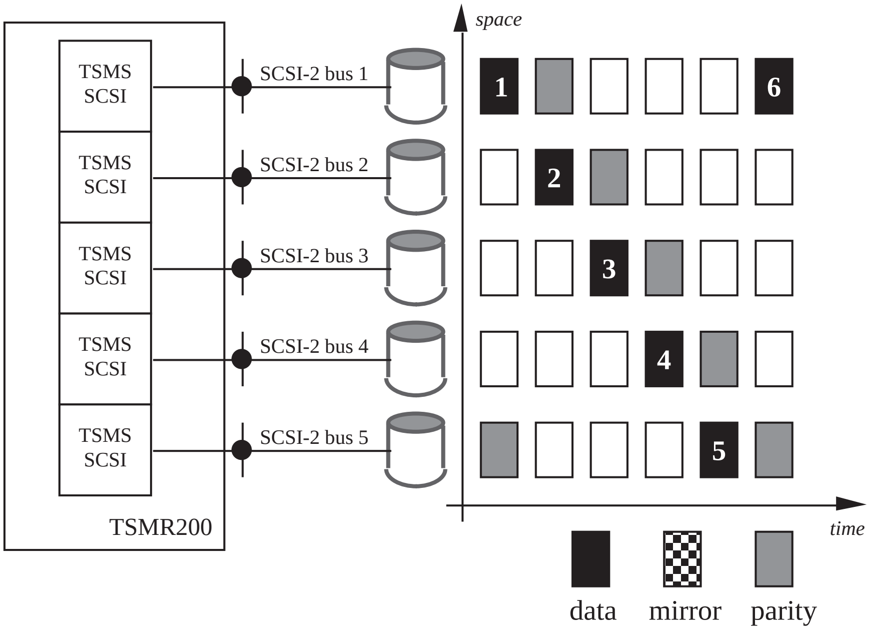


Figure 6.2: RAID Level 5

The advantages are that redundancy improves tolerance to disk faults and availability over RAID level 0, that, along with levels 3 and 4, this is the least expensive solution with the greatest storage capacity, and that other disks can be seeking and transferring data in parallel, giving improved throughput. The disadvantage is that every write involves a parity read-modify-write.

6.2.1.7 Multiple-Rank and Multiplex-Controller RAIDs

Most disk interconnection technologies allow more than one disk to be attached to each disk controller, which may then control disks from several ranks. When multiple RAID controllers are connected to the same disk array in duplex (see Figure 6.3), triplex or quadruplex configurations, this gives an increased tolerance to RAID controller faults.

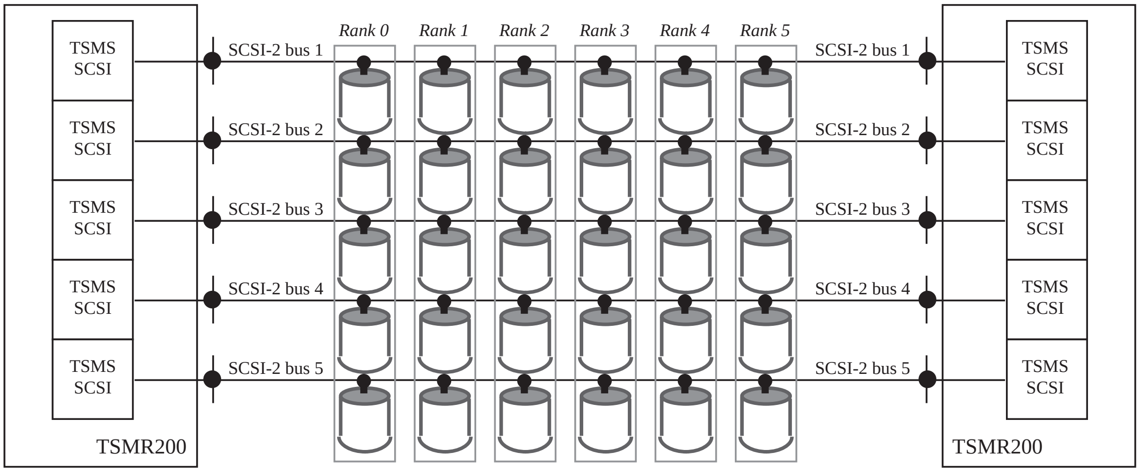


Figure 6.3: Multiple Rank RAID with Duplex Controllers

6.2.2 The First RAID Prototype

The first RAID prototype, RAID-I, was built at the University of California at Berkeley [Schultze 88]. It consisted of the following off-the-shelf components:

- **(a)** 1 Sun-4/280 with 128MB of memory.
- **(b)** 7 32-bit VME-SCSI Host/Bus Adapters.
- **(c)** 32 CDC Wren-IV 340MB 5.25 disks.
- **(d)** 1 Ethernet Interface.

The Berkeley RAID group discovered a hierarchy of bottlenecks which limited the overall performance of their system [E.K.Lee 90, Chervenak 90, Chervenak et al 91, Chervenak 91]. Performance was critically affected by a pathological interaction between the Sun-4/280's memory system and the Sprite operating system. These problems were addressed in their second generation RAID-II prototype [E.K.Lee 91], and strongly influenced the approach taken below.

6.3 Stable Disk

A special disk subsystem has been proposed for the FASST architecture, initially intended for use in a Futurebus+ environment [Futurebus+ 94a], and subsequently retargeted for the EISA I/O bus [EISA 92] of a shared memory multiprocessor PC system manufactured by Corollary Inc. [Corollary 92]. The basic principles, however, are not dependent upon any specific host bus features.

The Corollary machine is based on a dual bus architecture. An EISA bus provides support for normal PC I/O cards while a proprietary Extended C-Bus (EC-Bus) provides access to shared global memory and implements a proprietary multiprocessing cache coherency protocol. The system is typically populated with a base CPU board, one asymmetric CPU board, and four symmetric CPU boards. Asymmetric and symmetric boards are distinguished by the fact that only symmetric boards have access to both the EC bus and EISA bus. All boards have Intel i486/DX2 66MHz processors, and 1MB of second level cache. The system typically might have 64MB of global shared memory.

The special disk subsystem, the Stable Disk [Coghlan et al 92a], has been developed by Tolsys Ltd., a campus company of Trinity College, Dublin. It consists of one or more RAID Controllers, one or more companion Stable Memories, plus a large array of disks (see Figures 6.4 and 6.5).

The Stable Disk is principally designed for a *streaming* environment, where data flows to and from disk in large blocks, such as would be the case when using RAID 3 with a Log Structured Filesystem (LFS) [Ousterhout et al 88, Ousterhout et al 89]. It is also designed to be integrated into the FASST recovery protocol.

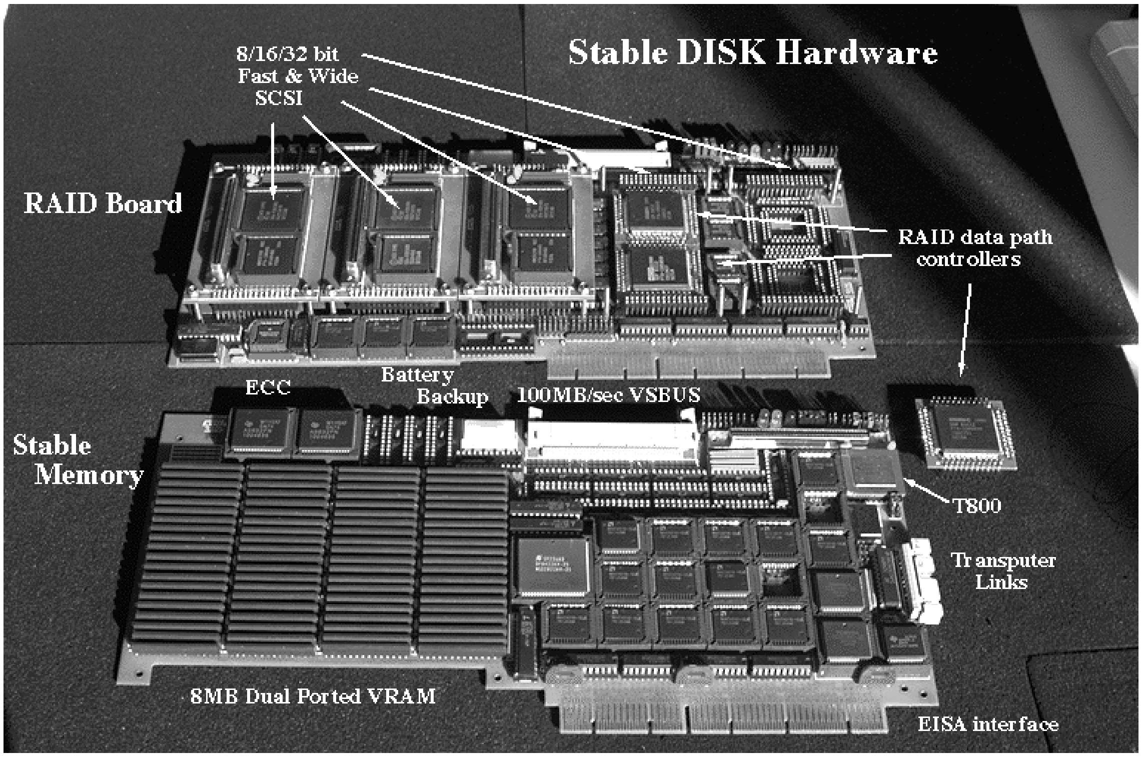


Figure 6.4: Stable Disk : RAID Controller and Stable Memory

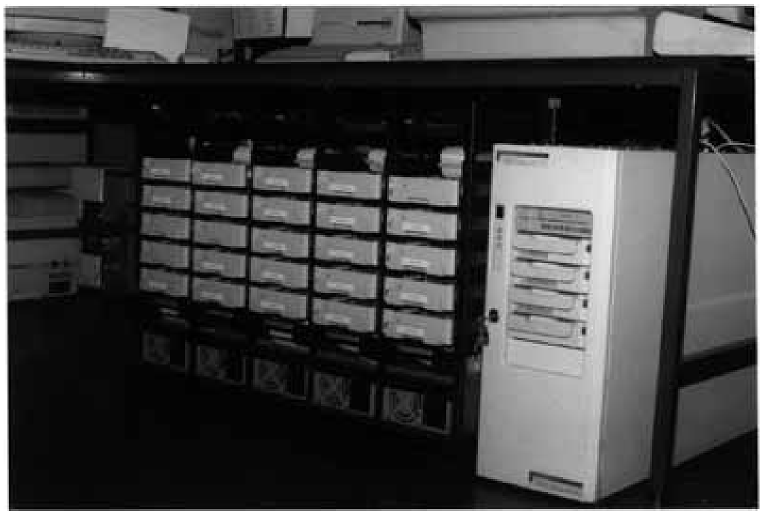


Figure 6.5: Stable Disk : RAID Chassis and Corollary host

6.4 RAID Controller for the Stable Disk

Figure 6.6 illustrates the low level architecture of the RAID Controller, which includes:

- **(a)** Plug-in SCSI modules that allow for 1 to 5 SCSI-2 ports.
- **(b)** Plug-in array modules that allow for 8bit, 16bit or 32bit RAID strings.
- **(c)** Hardware support for RAID levels 0, 1, 3 and 5.
- **(d)** An EISA host bus interface that supports voting for nMR operation.
- **(e)** A private 100MB/s VSBus interface.

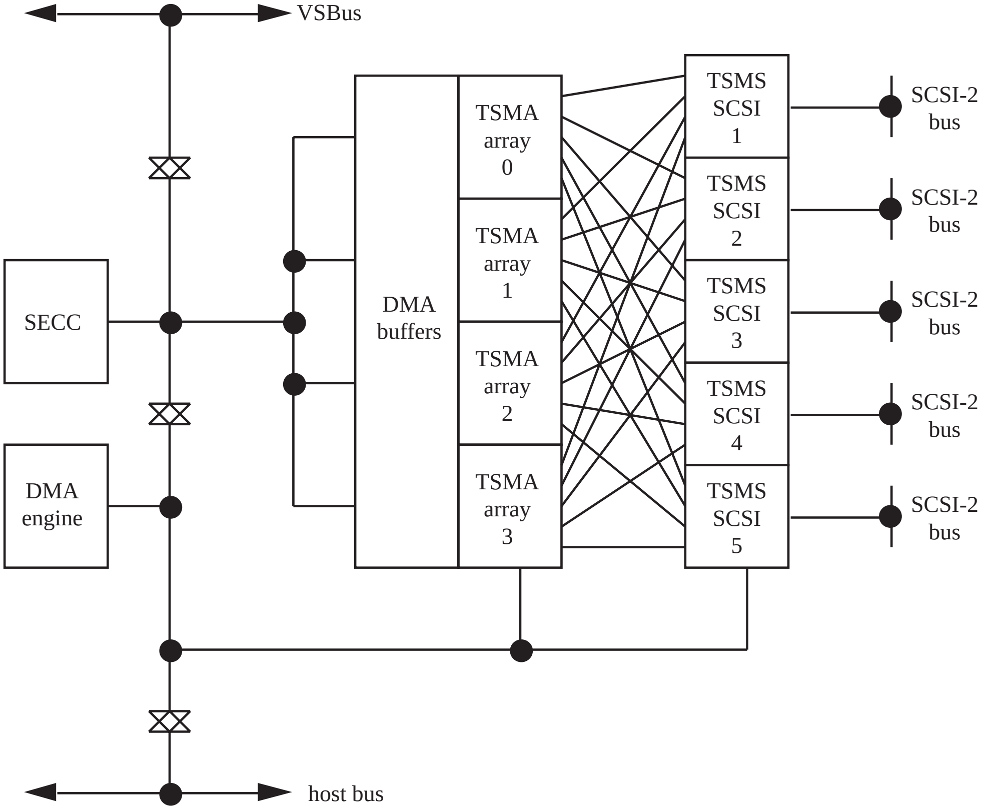


Figure 6.6: RAID Controller Block Diagram

This RAID Controller is designed to allow high-performance fault-tolerant disk subsystems to be constructed that have up to five 32bit RAID strings, each with up to seven ranks, and with multiplex controllers. In concept, it comprises a host bus programming interface, an external VSBus interface for stream-like data transfers independent of the host bus, an array data path that provides hardware support for RAID operations, and a set of disk controllers for interfacing to disks.

The logic of the array and disk interfaces is separated. An array module implements a byte-wide slice of the array data path; up to four of these may be installed. A SCSI disk controller module implements a single disk interface, five of which may be installed. This flexibility allows non-array disk subsystems, or array subsystems that do not need the usual hardware support, to be constructed.

A SCSI disk controller module contains a 16bit SCSI-2 disk controller [NCR53C916a, NCR53C916b] plus a SCSI bus extender [NCR53C932] that expands the SCSI bus width to 32bits. These are very comprehensive devices, which support a wide range of SCSI-2 operating modes. They may be set up for asynchronous, synchronous, fast or wide SCSI bus protocols. Both low-level commands and high-level macro-commands are accepted. The latter execute instructions to perform multiple-phase SCSI-2 sequences including phase and information decision making, program flow control, status reporting, error recovery and progress indication. These eliminate many time consuming interrupts and thus significantly reduce the overheads of managing the SCSI-2 bus. Nonetheless, performance still suffers from close interaction between the disk controller and array data path actions, and therefore would benefit from their decoupling via some form of randomly accessible buffering.

An array module implements an 8bit slice of the array data path, using a device [NCR53C920a, NCR53C920b] that provides hardware support for RAID levels 0, 1, 3 and 5. These modules are not absolutely necessary for an array subsystem, but if configured without them, performance suffers badly. To construct a minimal array configuration with hardware support, at least one array data path module must be installed. This will allow an array with 8bit SCSI interfaces. For 16bit interfaces, two array data path modules must be installed, whilst for 32bit interfaces all four modules must be installed.

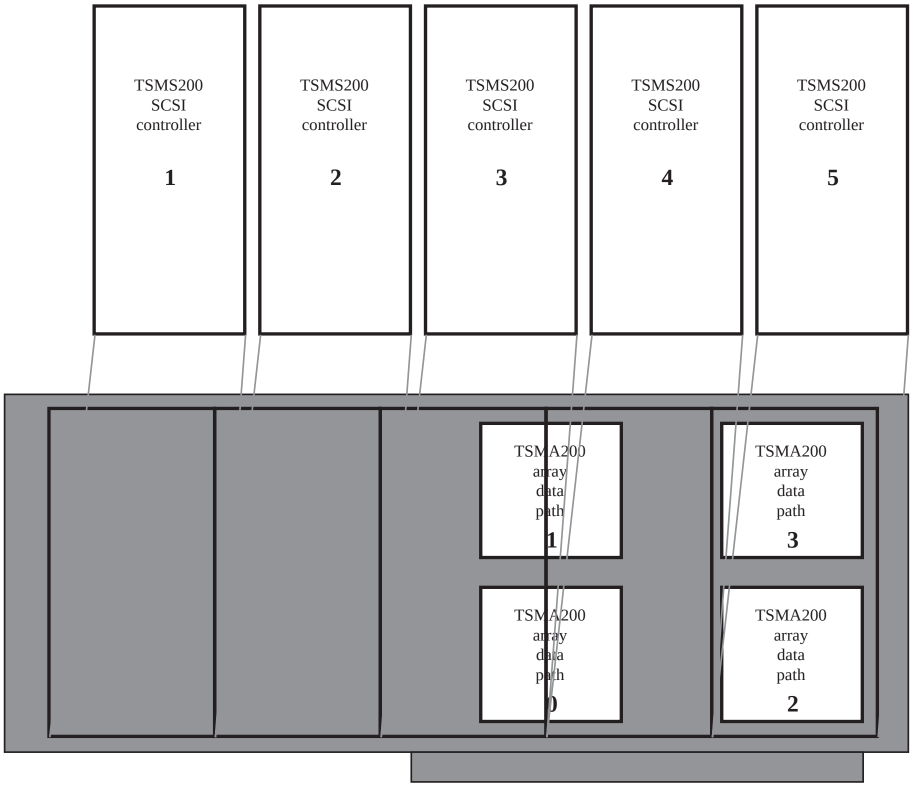


Figure 6.7: Layout of modules on the RAID Controller

Particular attention has been paid to provisions for redundancy. A single RAID Controller is clearly a single point of failure, although it may well be reliable enough for certain applications. For fault-tolerant operation a number of controllers can operate together in a nMR fault-tolerant arrangement. For example, 3 controllers may work in a triple modular redundancy (TMR) arrangement. The host CPUs would broadcast data and commands to all replicas, and would read data from all replicas (a gross simplification, since the nMR does not extend to the SCSI interfaces). If a controller detected a difference between the data on the host bus and the data at the input to its output buffers, it would trap to an error handler. The faulty controller would isolate itself from the bus and indicate that it needed repair. A tri-state host bus such as EISA is not entirely suited to this arrangement (a wire-OR host bus such as the open-collector Futurebus+ would suit better), so the RAID Controller has a generic host bus interface allowing a single active master with multiple passive slaves, with role reversal if needed.

Each RAID Controller is a member of a nMR group of up to three controllers, which all respond to the same addresses and hence operate in parallel. Its operation is controlled by a number of memory mapped registers, and is symmetrically redundant in that all the registers can be accessed from the host bus and the VSBus.

The control registers are located in window slots at the top of the companion Stable Memory's address space. The top 64KB of each Stable Memory is reserved for memory mapped I/O, and a RAID controller occupies 4kB within this address space, located at a base address derived from *nMR_group_no* and *nMR_controller_no* straps.

Accessing the control registers within this region allows commands to be written to a single RAID Controller within a group or broadcast to all controllers in a group, or even all groups. Accesses from the host bus can be disabled if felt necessary; e.g. for the determination of which controller is faulty in a nMR group.

The companion Stable Memory is designed to be able to check on each access whether a processor is allowed to access or modify the relevant memory page. By locating the RAID Controller's control registers in slot windows within the Stable Memory's address space, this protection mechanism is automatically extended to the controller whenever it is used in conjunction with the memory. Any accesses from the host bus will be checked by the memory's protection mechanism, and violations of that protection will be handled by that mechanism.

6.4.1 Information Flow

The array data path modules connect to a 128kB DMA buffer for read data, and another of the same size for write data, which allow burst transfers to or from the array data paths. These buffers are structured as FIFOs. The DMA buffers decouple the array DMA activity from any DMA activity on the host bus or VSBus, so that the two can proceed at independent rates, and may even occur concurrently. Typically for transfers to disk, for example, the transfer to disk of block J of data would be conducted via DMA out of the DMA buffers to the array concurrently with the transfer of block $J + 1$ of data from the VSBus into the DMA buffers.

Normally DMA is used to transfer data on the VSBus from/to the VSBus in/out of the DMA data buffers. However, a mode bit may be programmed to divert the DMA transfers so that they may be performed from/to the host bus. In either case the DMA width is fixed at 32bits and the *TRSHTOBUS* or *TRSHFRBUS* register is written to initiate the shift:

```
*TRSHTOBUS = count;   -- word count
```

When the DMA is diverted via the host bus, it may proceed under hardware control or via host bus memory mapped I/O accesses to *TRHOSTDMA*. In the first case the DMA proceeds automatically. In the latter case the DMA accesses must be programmed:

```
for (i=0; i<count; i++)
  *TRHOSTDMA = data[i];  -- loop doing DMA
```

DMA between the buffers and the array data path proceeds under hardware control. The *TRSHTOARRY* or *TRSHFRARRY* location is written to initiate the transfer:

```
*TRSHTOARRAY = count;   -- word count
```

While DMA can be performed via the host bus, it is intended to be done to or from Stable Memories via the VSBus. This bus is modelled on the DT-Connect II standard proposed by Data Translation Ltd., and is defined as an open interface specification, a 32bit extension to their earlier 16bit DT-Connect standard [DT-Connect, DT-Connect II]. Generally these busses are used for connecting data-acquisition, frame-grabber and image processing boards together by a ribbon cable, so that transfers of information between them need not involve the system bus. The VSBus does likewise. Figure 6.8 illustrates how a number of RAID Controllers and Stable Memories may be interconnected by the VSBus. Transfers along the VSBus take place at 100MB/s with negligible reduction in host bandwidth ($<$2%).

There can only be one master of the bus at any time, the others being either slaves or inactive. A master can communicate with more than one slave, in broadcast fashion. The master and slaves must be selected as such by the host via the host bus. There are no addresses on the bus, and data transfers occur sequentially (i.e. in burst mode). It is up to the master and slaves to handle the data as necessary. The control algorithm is as follows :

```
{
  set-up slaves
  start master
  wait for completion
}
```

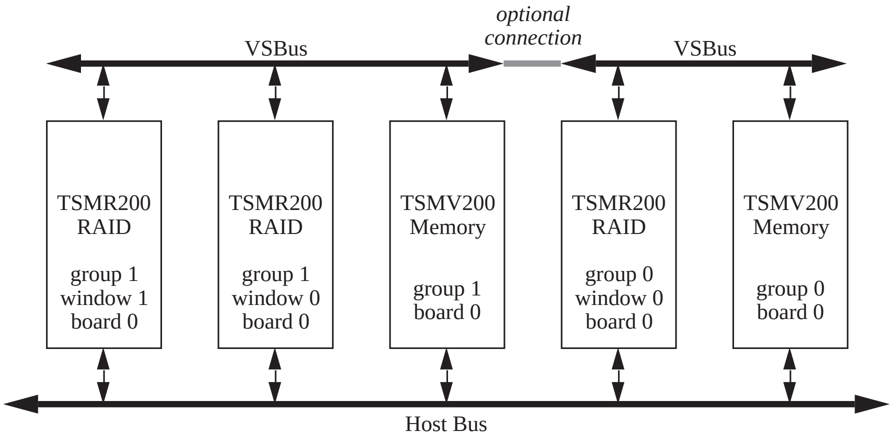


Figure 6.8: Interconnection of RAID Controllers and Stable Memories using the VSBus

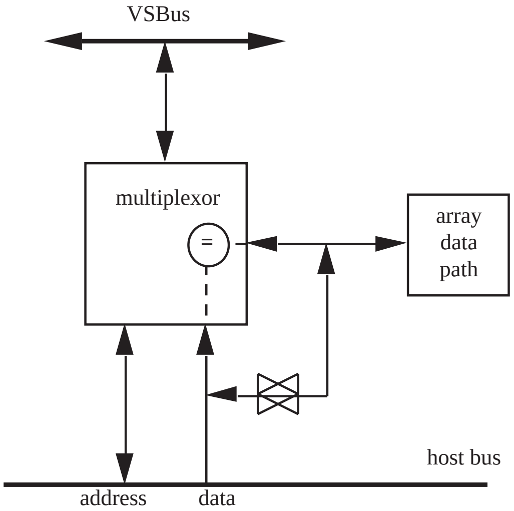


Figure 6.9: Bus Interface/Comparator for nMR Operation

6.4.2 nMR Operation

Both the RAID Controllers and the Stable Memories contain an interesting voting circuit for use in nMR fault-tolerant configurations. Groups of 2 or 3 RAID Controllers can be configured with the same group address; similarly for the Stable Memories. One unit must be set up as the nMR master and the others as nMR slaves.

For all write accesses (memory and host bus memory mapped I/O) all units in the group perform the writes. For memory read accesses all units in the group respond internally, but only the master drives the host data bus. Both master and slaves compare the host data with their own internal data. If a discrepancy is found the host bus error signal is activated before the host bus wait signal is released, an interrupt is registered and possibly forwarded to the VSBus or the host bus. The faulty unit must be identified and isolated. This can be performed by disabling the host bus buffers and testing each unit in turn. A new master may need to be selected.

For host bus memory mapped I/O read accesses all units in the group respond internally, but only the master drives the host data bus bits *d30-d0*. Both master and slaves compare these host data bus bits with their own internal data. If a discrepancy exists, data bit *d31* is driven low by the unit(s) to indicate disagreement (no error or event is signalled). This allows broadcall status reads where a CPU may busy-wait on *d31*. When *d31==1* all units have reached identical status. Obviously timeouts need to be used to limit the busy-wait duration. The special circuitry generating *d31* on each unit performs the following function:

```
MASTER:                          SLAVE:
  if (agreement & d31)               if (disagreement)
    d31 = 1;                           d31 = 0;
  else                               else
    tristate(d31);                     tristate(d31);
```

For host bus memory mapped I/O accesses, only *d0-d30* are checked. For memory accesses *d31* will be checked as well, so eventually a faulty *d31* will be detected on memory accesses anyway. However, even if this were not so:

(a) if *d31* should be 1 and it is 0, then a timeout will occur, so this is not a problem, since that unit would then be identified by diagnostics and its host interface disabled. Note that a 0 will override a 1 for both open-collector busses (eg. Futurebus+) and TTL busses (eg. EISA).

(b) if *d31* should be 0 and it is 1, then that unit is a nMR slave anyway, since only a slave activates *d31=0*. It is faulty, but does not affect the operation of the nMR master. If it is subsequently chosen as a nMR master, this fault will not affect its behaviour as a master.

6.5 Stable Memory for the Stable Disk

The second step in extending the ability to tolerate processor failures to secondary storage is to design the disk controller cache as a stable memory. The purpose of such a cache is to provide private memory for intelligent use by the disk controller. The system would be simpler without this, but in the absence of a private path to memory many disk interactions will unnecessarily consume system bus bandwidth. Unfortunately, too, the trend is to larger disk caches. If they were small then they might be managed within the recovery protocol as a special case. For larger disk caches it is more efficient if they are designed to have the same functionality as main memory (i.e. the SM).

The Stable Memory of Figure 6.10 is designed with this in mind, and also as an experimental testbed. It has the following interesting features:

(a) It includes 8MB (or 32MB) of ECC protected VRAM.

(b) It supports multiple checkpointing methods (intra-bank, interbank, log-mode and [optionally] instantaneous) that can checkpoint at rates from 100MB/s to 40GB/s.

(c) Protection logic is optionally supported, with per page access control & status update, and CPU-CPU centralised dependency tracking.

(d) It is managed by an embedded controller with cache, that includes serial links that may be used for data recovery.

(e) The EISA host bus interface supports voting for nMR operation.

(f) A private 100MB/s VSBus interface is also provided.

The Stable Memory is an intelligent memory system based on video-RAMs (VRAMs). VRAMs are dual ported memories having a parallel and a serial port. The host CPUs can access the VRAMs as normal memory from a host bus using the VRAM's parallel port. The VRAM serial ports are interconnected via a local bus which also is connected via buffers to the external VSBus, so that it can be used as a private path for high speed I/O (principally disk I/O).

The primary support for the recovery protocol is through checkpointing and rollback. Memory can be dynamically partitioned into normal and stable 4KB pages. For a stable page, two pages are allocated. One page is active and the other is the backup. After a system failure, data stored in a stable page is recoverable from the backup copy.

A number of checkpointing methods are supported. Firstly, there are several checkpointing methods based on physically copying between the active and backup pages [Coghlan et al 89, Coghlan et al 91, Coghlan et al 93], the speed of which is greatly enhanced by the use of VRAMs. Alternatively, log-mode checkpointing maintains a log of the addresses and original values of all modified locations, which may be replayed in reverse order to return to the previous checkpointed state. Finally, switch-mode checkpointing [Coghlan et al 92b] reverses the active and backup roles by simply writing to a control register.

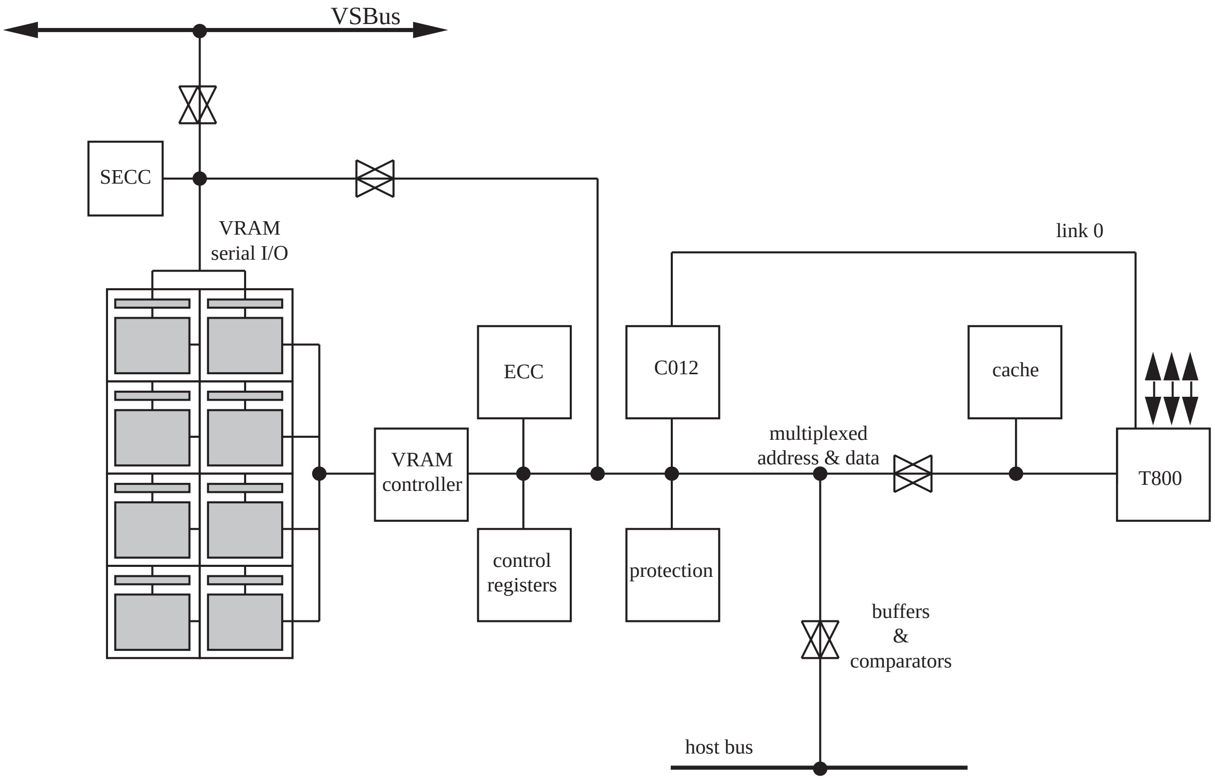


Figure 6.10: Stable Memory Block Diagram

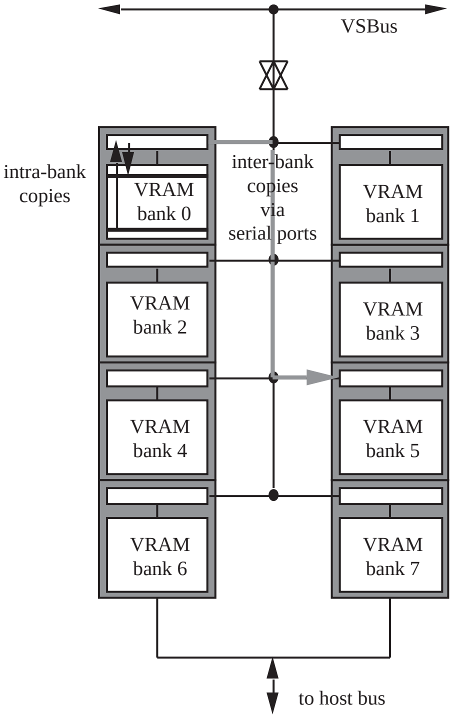


Figure 6.11: Organization of VRAM Banks in the Stable Memory

As with the RAID Controller, a number of Stable Memories can operate together in a nMR fault-tolerant arrangement, utilising the same mechanisms and protocols. If the bus itself turns out to be the point of failure, then the data held in the VRAMs may be transferred to an alternate (perhaps clustered) system over high speed serial links.

The Stable Memory has 8MBytes (or 32MBytes) of ECC protected memory organised as 8 banks of 10 VRAMs each (see Figure 6.11), and hence each Stable Memory occupies that quota of address space. Again as with the RAID Controller, each Stable Memory is a member of a nMR group of up to three memories, which all respond to the same addresses and hence operate in parallel. The Stable Memory is located at a base address derived from its *nMR_group_no*, and successive 4KB memory pages reside in successive banks. The top 64KB of its address space is reserved for memory mapped I/O, and its operations are controlled by a set of memory mapped control registers located within this region at a base address derived from *nMR_group_no* and *nMR_memory_no* straps. Again, the addressing scheme allows accesses to or from an individual Stable Memory, broadcast to all memories within a group, or broadcast to all groups in a system.

The Stable Memory has an embedded *T800* transputer [InMOS 88] controller. A Stable Memory's address space is accessible from both the host bus and the *T800* so that its operation can be controlled by either. Access via the host bus can be enabled/disabled by writing to the appropriate control registers (*TVBUFEN*, *TVMEMEN* & *TVMMIOEN*), so that the *T800* can be totally in control. It is also possible for *T800 firmware* in VRAM to execute high level commands on behalf of the host CPUs. To reduce contention for VRAM access, a 32KB write-through cache is provided for the transputer. The *T800* transputer links may be used for remote recovery of Stable Memory state. A *C012* [InMOS 88] is also provided for compatibility with the Transputer Development System (TDS) [InMOS 90].

6.5.1 Intra-bank Checkpointing

A VRAM is a DRAM with the addition of a serial access register and a serial port (see Figure 6.12). In addition to the standard DRAM cycles, transfer and shift operations can be performed on the serial access register and the serial port. A transfer operation copies a complete memory row (typically 512 bits) between the memory array and the serial access register or vice-versa. A shift operation selects the next data bit in the serial access register to be output on the serial I/O port (or vice versa). These additional cycles can be used to copy data efficiently. Copying data within a VRAM (device) is called an intra-bank transfer and copying between different VRAMs (devices) is called an inter-bank transfer.

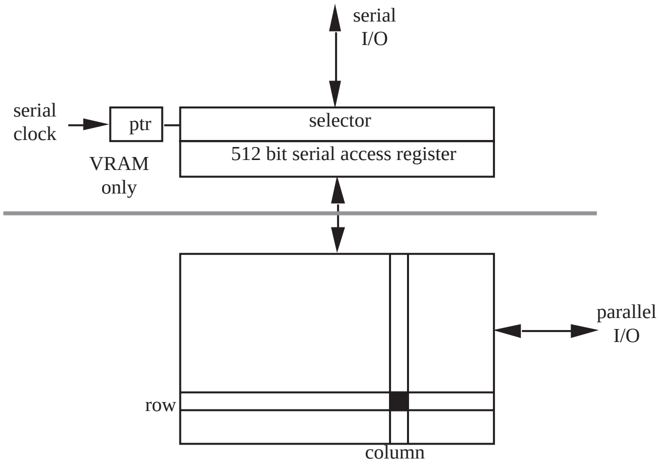
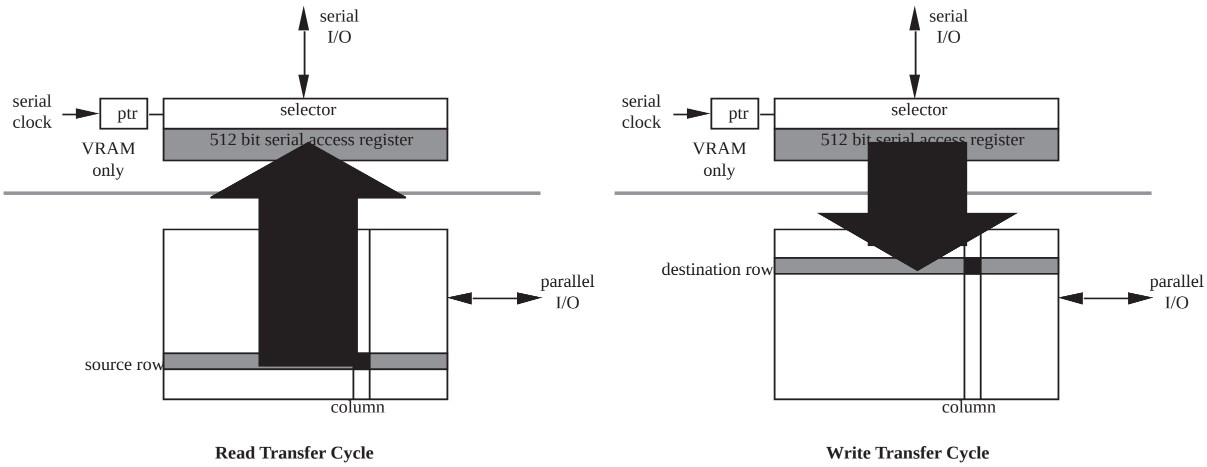


Figure 6.12: Simplified VRAM Block Diagram

An intra-bank transfer within a VRAM requires a read transfer cycle followed by a write transfer cycle. Since each bank of the memory is 32 bits wide (excluding ECC), there are 32 x 512 bit serial access registers so 2kB of data is transferred in two memory cycles (approximately 400ns), as shown in Figure 6.13. Only complete rows can be transferred using intra-bank transfers.



Read Transfer Cycle

Write Transfer Cycle

Figure 6.13: Intra-bank Transfer

If the two pages of a stable page are allocated in the same bank (VRAM device) then *checkpointing* can be performed using VRAM intra-bank transfer cycles. A read transfer cycle followed by a write transfer cycle can copy 2kBytes of data. A transfer masked cycle is also available making it possible for all, or a subset, of the 8 VRAM banks to perform transfer cycles in parallel. The banks are specified by an 8bit stability mask (a bit for each bank) from the protection logic or as an immediate mask from the data field of the transfer cycle. The stability mask is useful for global checkpointing where some of memory is non stable (eg. code areas). Given that transfer cycles can take place in parallel over all banks, only 100μs (256 rows x 200ns x 2) is needed to checkpoint all of memory using intra-bank transfers. The checkpointing time depends only on the organisation of a bank, and not on the number of banks per Stable Memory or the number of Stable Memories in a system.

A read transfer cycle is performed by writing to register *TVRXFR* with the data being the ”transfer address”. A write transfer cycle is performed by writing to *TVWXFR*. Note that the low order 11 bits of the ”transfer address” are not used. Hence an intra-bank transfer is performed as follows:

```
*TVRXFR = src;
*TVWXFR = dst;
```

To perform simultaneous transfer cycles on all or a sub-set of the 8 VRAM banks by executing a ”transfer masked” operation, the low order 9 bits of the ”transfer address” is split into a single bit *ALL* field and an 8 bit *MASK* field. If *ALL=0*, the *MASK* bits are used to enable/disable the transfer cycle on each of the 8 banks. A transfer of all 8 banks simultaneously from source to destination is performed as follows (assuming the source (*src*) and destination (*dst*) addresses are aligned on 2KB boundaries):

```
*TVRXFRM = (src | 0xff);
*TVWXFRM = (dst | 0xff);
```

For global checkpointing, non-stable pages must not be checkpointed. If present, the protection logic outputs a per-page 8bit stability mask on each access, and if *ALL=1* the stability mask is used, as above, to enable transfer cycles on each VRAM bank. To avoid replication of the stability mask, masked transfers need to be aligned on 8 page boundaries. A transfer of all 8 banks which depends on the associated stability mask is performed as follows:

```
*TVRXFRM = (src | ALL);
*TVWXFRM = (dst | ALL);
```

Pseudo-write transfers are used to turn the VRAM serial I/O buffers to input mode without transferring the data from the serial access register to a memory row. A pseudo-write transfer is performed by:

```
*TVPWXFR = dst;
```

6.5.2 Inter-bank Checkpointing

An inter-bank transfer between different VRAM banks requires the following steps:

(1) read transfer of source row into VRAM serial access register.

(2) shift the data between the source & destination VRAM serial access registers using the serial bus.

(3) write transfer from VRAM serial access register to destination row.

A complete row is transferred with 2 memory cycles and 512 serial clock cycles. The memory is 32bits wide and the serial port is also 32bits wide. Since the serial clock operates at 25MHz, this corresponds to 2kB transferred in 20.88μs (100MB/s). An important point is that while the VRAM parallel port is occupied for the two transfer cycles, lasting approximately 400ns, it remains available for the next 20.48μs, resulting in negligible reduction ($<$2%) in host bus bandwidth during inter-bank transfers. Note that partial rows can be transferred using inter-bank transfers.

Although the VSBus is used to interconnect Stable Memories and RAID Controllers, it also is used for inter-bank transfers when the source and destination banks are on different memories. The transfer operation is similar to the inter-bank transfer described above, except the data is enabled onto the external VSBus. Data is transferred between a master and a number of slaves. As with local inter-bank transfers, transfers along the VSBus take place at 100MB/s with negligible reduction in host bandwidth.

If the two pages of a stable page are allocated in different banks then *checkpointing* can be performed by using inter-bank transfers to copy data from the active page to its backup. Inter-bank checkpointing involves the same three steps outlined above, and thus takes place at 100MB/s with negligible reduction in host bandwidth.

In each case the data must be shifted in or out of the serial access registers. The *SHIFT* register is written to initiate the shift. The shift count and source bank are encoded in the data value. A typical inter-bank transfer sequence would be performed as follows:

```
*TVPWXFR = dst;
*TVRXFR = src;
*TVSHIFT = ((src & 0x3800) | 512); -- 512 bit serial
                                 ;    access registers
*TVWXFR = dst;
*TVRXFR = src;
*TVSHIFT = ((src & 0x3800) | 512); -- 512 bit serial
                                 ;    access registers
*TVWXFR = dst;
```

Data can be shifted between units by writing to *TVSHTOSBUS* or *TVSHFMSBUS* depending on whether data is being transmitted or received from the VSBus . To send data to a slave via the VSBus:

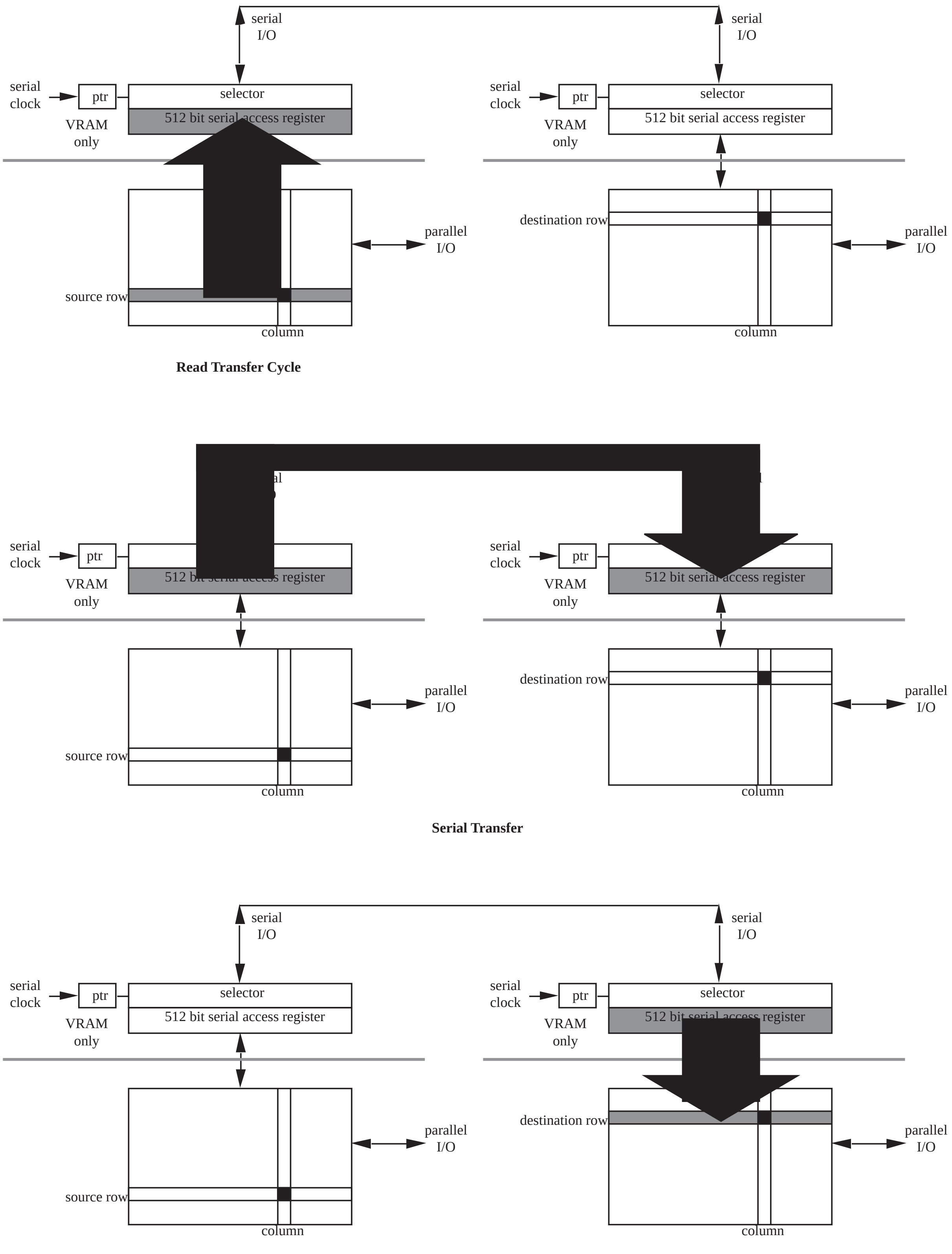
```
*TVSHTOSBUS = ((src & 0x3800) | 512); -- 512 bit serial
                                    ;    access registers
```

In this mode only one Stable Memory (in a nMR group) drives the VSBus. The destination for the data is determined by which memories have been set up as VSBus slaves. The current value of the shift count can be obtained by reading *TVSHIFT*.

6.5.3 Log-mode Checkpointing

Log-mode checkpointing involves recording a log of memory updates, similar to SM's use of its *update* list (see section 4.1.1). On every write, the original value of the location together with its physical address are stored in a log. In this implementation the log is stored in a reserved area of VRAM (see Figure 6.15). On rollback, memory can be returned to its previously checkpointed state by replaying the log in reverse order.

Log-mode checkpointing is enabled by setting *TVMODE* to *LOGMODE*. The starting address of the log is set by writing to the register at *TVLOGADDR*. On a checkpoint, *TVLOGADDR* can be reset to this value (only one log), or set to a different value (more than one log). On a write, the original value of the location together with its address are saved in the log, the value of *TVLOGADDR* is incremented by 8 (bytes) and then the memory location itself is updated. The advantage of log-mode checkpointing is that it uses less memory, but at the cost of slower writes (a factor of 3 in this implementation).



Read Transfer Cycle

Serial Transfer

Write Transfer Cycle

Figure 6.14: Inter-bank Transfer

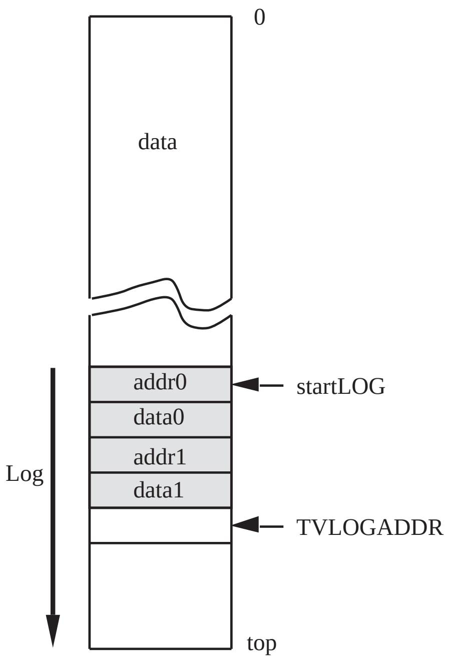


Figure 6.15: One Possible VRAM Organisation for Log-mode Checkpointing

6.5.4 Switch-mode Checkpointing

Assume that the two locations of a stable location are in different memory banks. One acts as the active location, the other as the backup. The idea of switch-mode checkpointing is to reverse the active and backup roles by writing to a single memory mapped register *TVSWITCH*. This gives effectively instantaneous checkpointing. Switch-mode checkpointing is enabled by setting *TVMODE* to *SWITCH*.

An active/backup location has two status bits associated with it. The *M* (modified) bit indicates whether the location has been modified since the last checkpoint. Immediately after a checkpoint, the active and backup locations are the same until the active location is modified. The *AB* bit indicates which is the active bank (i.e. most up to date). The mechanism assumes that the *M* bits are stored in a memory which can be cleared (reset) by writing to register *TVSWITCH*. Circuit operation is explained by the state transition diagram and truth table for a single location, as shown in Figure 6.16 and Table 6.2.

M	**AB**	**read**	**write**	**checkpoint**	**rollback**
0	0	read bank 0	(bank 0 → bank 1) write to bank 1 state → 11	state → 00	state → 00
1	1	read bank 1	write to bank 1	state → 01	state → 00
0	1	read bank 1	(bank 1 → bank 0) write to bank 0 state → 10	state → 01	state → 01
1	0	read bank 0	write to bank 0	state → 00	state → 01

Table 6.2: Switch-mode Checkpointing Truth Table for a Stable Location

If *M = 0* (unmodified) and *AB = 0* (state 00) then reads are from bank 0 (assume this is the initial state of all locations) until a write occurs which is directed to bank 1, while the state is simultaneously set to 11. Subsequent reads are now from bank 1 and the *M* bit has been set to indicate that the location has been modified. If a failure occurs at this point, rollback is achieved by simply reverting back to state 00. On a checkpoint the *M* bit is cleared to move into state 01. Checkpointing in states 01 (and 00) leaves the state unchanged as the the backup and active locations are in the same bank.

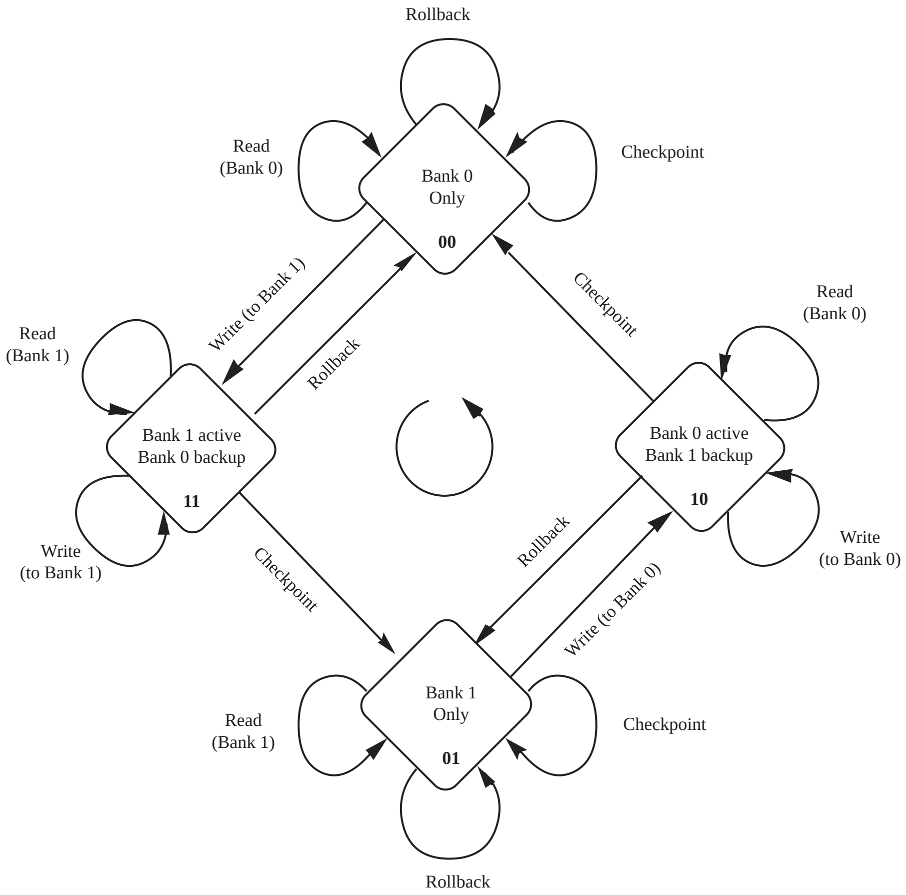


Figure 6.16: Switch-mode Checkpointing State Transition Diagram for a Stable Location

Notice that checkpointing is accomplished by simply writing to *TVSWITCH*, but rollback (a hopefully less frequent event) requires the status array to be selectively updated. If status information has a larger granularity than one location then copy-on-write operations are needed. This is the case for the Stable Memory.

6.5.5 Protection Logic

The protection logic is designed to be implemented on an interchangeable module that plugs into the Stable Memory. It checks if a CPU has the necessary rights to access a particular memory page, updates per page status information and maintains a CPU-CPU dependency matrix. Protected window slots are provided in the address space for access to companion RAID Controllers. When a companion controller is accessed using the host bus the protection is provided by the Stable Memory. It is also possible to access the RAID Controllers in the same protected manner via the VSBus. The protection logic deals with the following issues:

(a) Checking whether a process is allowed to access a particular memory page. The protection logic checks each CPU access against a set of access rights. These are a function of the currently executing process and the memory address. A *CPU* number is bound to a process index (*PIX*) by the operating system scheduler on a context switch. Each process, thread or interrupt handler should be given a separate process index. CPUs are identified by host bus user-defined signals or by the higher-order host bus addresses. Each bus master (including DMA controllers) should be given a separate *CPU* number.

(b) Maintenance of per page status information. Per page status information is obtained by indexing into a status RAM (*STSRAM*). Each *STSRAM* entry has a number of fields such as a referenced bit, a modified bit and a stability mask which are updated appropriately on each access.

(c) Maintenance of a CPU-CPU dependency matrix. A dependency matrix (*DM*) tracks read-write and write-write dependencies between CPUs. The *DM* is used to make sure that dependent processes are checkpointed (or rolled-back) together. The dependency matrix is an N x N bit matrix, where N is the number of CPUs. If CPU_i is dependent on CPU_j, then *DM[*CPU_i*,* CPU_j*]* will be set.

The protection mechanism can be thought of as executing over 3 phases, as shown in Figure 6.17, although these phases are not necessarily reflected directly in the hardware implementation:

(1) The *CPU* number is used to fetch the process index *PIX* (not the same as the operating system *PID*) from the *PIXRAM*. Simultaneously the page number is used to index into the *STSRAM* to obtain the per-page status information (*STS*) and the *CPU* number of the last modifier (*LM*) of the page.

(2) The *TAGRAM* cache is searched for the *PIX* and the page number to obtain the access rights which the process has to the page. If there is a cache miss, higher level software should fill the *TAGRAM* cache line appropriately. At the end of phase 2 the access rights are available. Simultaneously the *CPU* number and the *LM* are used to access the dependency matrix.

(3) The *STSRAM* and dependency matrix are updated appropriately.

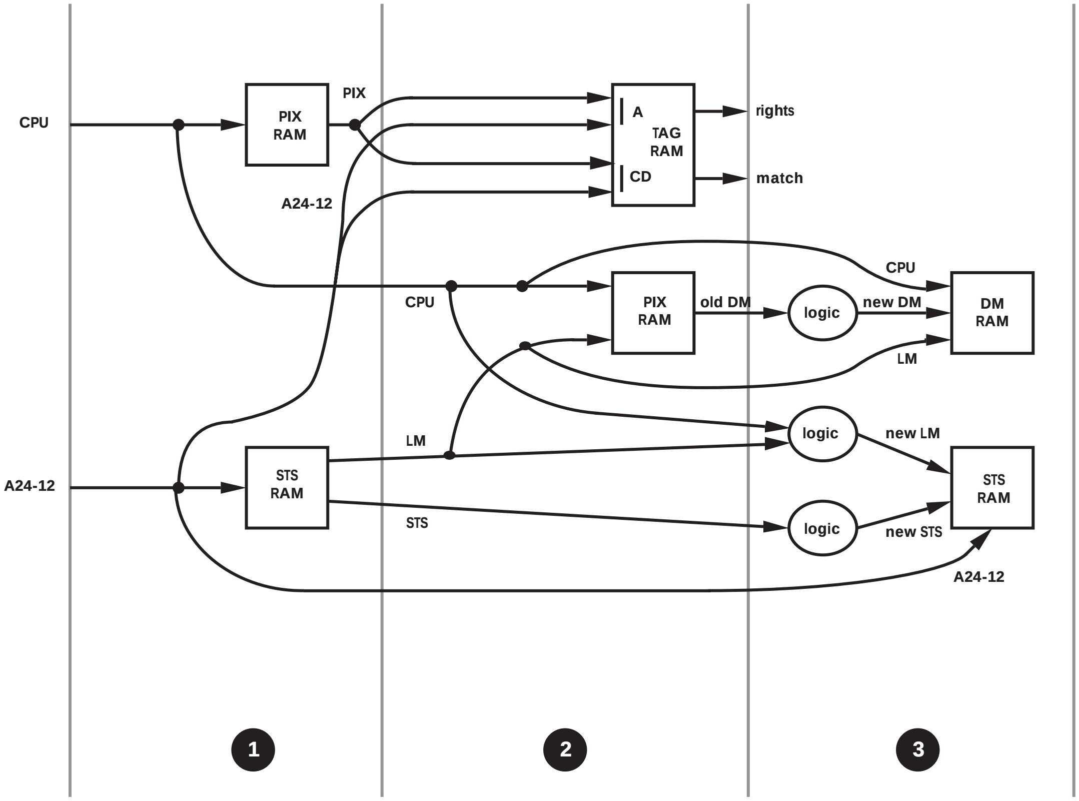


Figure 6.17: Protection Logic Operation

The page-based protection is very much in excess of what is required for the FASST recovery protocol, which needs just the dependency information. The *DM* values are 1bit entries in a matrix, that indicate the dependencies between the current process and the last modifier (last owner) of the page that is being accessed. The matrix rows are indexed by the *CPU* number. The matrix columns are indexed by the last modifier *LM*, which is updated on a write with the current *CPU* number.

For read-write dependencies, the current process is dependent on the last modifier of the page that is being accessed. This requires that the entry *DM[CPU, LM]* be set. For write-write dependencies, the last modifier of the page that is being accessed becomes dependent upon the current process. This requires that both the entries *DM[CPU, LM]* and *DM[LM, CPU]* be set.

6.6 Integration into the FASST Recovery Protocol

The protection logic allows the Stable Disk (SD), consisting of RAID Controllers and Stable Memories, to take part in the FASST Recovery Protocol described in [Morin et al 92] and Chapter 4. This section outlines how this is done. The protocol is extended as necessary, with the SM definitions of Chapter 4 abbreviated and the extra SD definitions in bolded italics. The recovery commands are in general emulated by software or firmware.

Each SD has the logical structure shown in Figure 6.18, and is composed of a single array of memory cells, grouped in pages :

```
type             t_bank = array[0..bank_size-1] of block ;
t_bank          Bank1, Bank2 ;

type             SD_mem = array[0..SD_mem_size-1] of page ;
SD_mem          SD_Memory ;
```

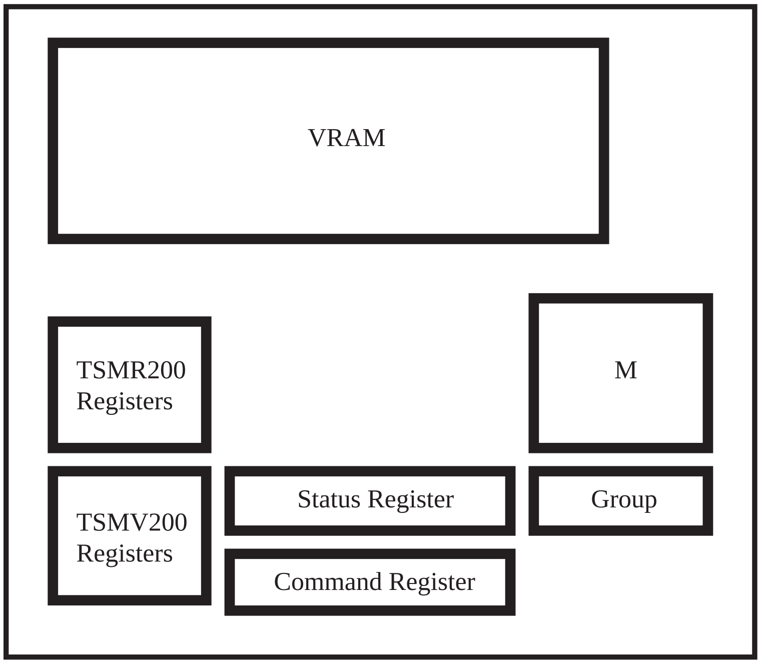


Figure 6.18: Stable Disk logical structure

The way that this memory is used depends upon the checkpointing mode in use. FASST uses selective rather than global checkpointing, so the intra-bank checkpointing mode is most natural. This assumes each page is paired with another to form a single stable page :

```
type             SD_t_bank = array[0..(SD_mem_size/2)-1] of page ;
SD_t_bank       SD_Bank1, SD_Bank2 ;
```

As for the SM, space is divided into a set of contiguous areas of identical size, with a current value in *bank1* and a backup value in *bank2*, but for the SD the granularity is a page rather than a block. SD read and write commands refer to SD memory cells, but the SD records dependencies at page granularity.

The following information is stored for each page :

```
type             t_vector_elt = record
   :
t_vector        Vector ;

type             SD_t_vector_elt = record
                                        t_LM LM ;
                                        t_status status ;
```

```
      ... additional information ...
                                   end ;
type SD_t_vector  array[0..page_nb-1] of SD_t_vector_elt ;
SD_t_vector       SD_vector '
```

The *LM* field contains the identity of the last modifier of the page, and is analogous to the *owner* field in the SM. The SD dependency matrix *DM* is analogous to the matrix *M* of the SM, also being updated during read and write operations, and henceforth will be referred to synonomously. It records dependencies between processors (or any other bus masters) that share memory blocks, and hence is an n x n Boolean matrix where n is the maximum number of processors. A matrix item $M(i, j)$ is set to true to indicate that bus master P_i is dependent on P_i.

Once a dependency group is computed by a processor it is stored in each SM and SD in the *group* field. Unlike the SM, the SD does not maintain an *update* list (although if this is essential it may be obtained by selecting log-mode checkpointing, at the expense of longer memory access times).

As in Chapter 4 the following algorithms are in psuedo-C, with *false* represented by 0 and *true* represented by a non-zero value.

6.6.1 Dependency management

The SD distinguishes between two types of dependencies:

(a) read after write dependency.

(b) write after write dependency.

In the FASST recovery protocol, dependencies are recorded by SM boards when processors access memory blocks and when necessary due to cache coherency actions. The SD performs the same recording but at page granularity. Since the SD memory is disjoint from the SM, this only affects the recovery protocol granularity - the algorithms remain valid.

Assume P_i is independent from all other processors. When P_i commits then firstly its cache is flushed to *bank1*, then secondly all the pages it has modified since its last commit point are copied from *bank1* to *bank2*. When processor P_i rolls back then all the pages it has modified since its last commit point are copied from *bank2* to *bank1* and its cache is invalidated. To detect which pages have been modified by P_i, each time a page is written by a processor, the processor identity is recorded with the page in the *LM* (last modifier) field.

6.6.1.1 Read after write dependency

Consider a processor P_i writes to a page B. Later the processor P_j reads the same page B. If P_j commits at time t then the last modifier of B, P_i, must also commit due to the dependency with P_j (otherwise a subsequent rollback of P_i would imply that P_j would have read a value of B which has never been written, so leading to an inconsistent state).

Symmetrically, if the last modifier of B, P_i, rolls back at time t then the reader of B, P_j, must also rollback due to the dependency with P_i (otherwise it posesses a value of B which has never been written to). Because of indeterministic behaviour of a system, nothing guarantees that after rollback P_i will write into B the same value as the one it wrote before rollback.

In summary, when P_j reads a page previously modified by P_i then P_j is said to be dependent on P_i ($P_i \leftarrow P_j$). A commit of P_j implies a commit of P_i and a rollback of P_i implies a rollback of P_j .

6.6.1.2 Write after write dependency

Consider a processor P_i writes to a page B. Later the processor P_j writes the same page B. If P_i commits at time t then the last modifier of B, P_j, must also commit due to the dependency with P_i (otherwise a subsequent rollback of P_j would imply that B is restored with a value which has never been written, so leading to an inconsistent state). Commitment of P_i implies that the value written by P_j is copied from *bank1* to *bank2*.

Symmetrically, if the last modifier of B, P_j, rolls back at time t then P_i must also rollback due to the dependency with P_j (otherwise it posesses a value of B which is different to that it wrote).

In summary, when P_j writes a page previously modified by P_i then P_i is said to be dependent on P_j ($P_i \rightarrow P_j$). A commit of P_i implies a commit of P_j and a rollback of P_j implies a rollback of P_i.

6.6.2 Synchronisation

6.6.2.1 Assumptions

```
int              p_nb ;                  /* number of processors        */
int              sd_nb ;                 /* number of stable devices,   */
int              active_p[0..p_nb-1] ;   /* array indicating which      */
                                          /* are active (valid)          */
                                          /* processors                  */
                              /* active[i] == 1 : processor i active      */
                              /* active[i] == 0 : processor i inactive    */


int              SM_nb ;                 /* number of SM boards         */
int              SD_nb ;                 /* number of SD subsystems,    */
sd_nb = SM_nb + SD_nb ;
```

6.6.2.2 Processor synchronisation states

The synchronization registers are in the processor boards, and the processor synchronization mechanisms are not affected by the presence of the SD. The processor automaton states *normal*, *stopping*, *stopped*, *recovery*, *restart*, *atomic_operation*, *waiting* and *failure* are as described in Chapter 4. The only proviso is that the initiator remains in *recovery* state until either SMs and SDs communicate *copy_done* or *timer(3)* runs over.

6.6.2.3 Stable device synchronization states

As for the SM, synchronization between processors and SDs is implemented with the status and command registers of the SDs. However, these registers are emulated in software or in the TSMV200 transputer firmware. The emulated command and status register values are stored in fixed non recoverable (non-stable) pages in the SD memory :

```
/* synchronization variables - one per stable device */
type             t_SM_synchro = array [0..SM_nb-1] of sd_synchro_state ;
t_SM_synchro     SM_state ;
type             t_SD_synchro = array [0..SD_nb-1] of sd_synchro_state ;
t_SD_synchro     SD_state ;
```

Apart from this emulation the SD is just another stable device :

```
/* stable device synchronization states */
type             sd_synchro_state = (normal,
                                     ready,
                                     commit_copy,
                                     rollback_copy,
                                     failure ) ;
```

The SD emulates the SM automaton as presented in Chapter 4. The emulation is performed in software or in the TSMV200 transputer firmware. The only difference is that for most failures the SD can return from the *failure* state to the *commit* or *rollback* state, as described in the discussion of TSMx200 exceptions errors in [Coghlan et al 92a].

6.6.2.4 Time-out protection

Timeouts are implemented with the `Wait` primitive. This primitive only needs to be modified to function with the emulated SD synchronization registers, which are in memory mapped pages in the SD memory rather than in CSR registers as in the SM :

```
Wait (condition, nb, t_max)
{
      timer_expiration = false ;
            :
            :
                  case all_SM_ready :
                        /* all SMs must be in the ready state  */
                        one_not_ready = (SM_state[i] != ready) ;

                  case all_SM_copy_done :
                        /* all SMs must be in the normal state */
                        one_not_ready = (SM_state[i] != copy_done) ;

                  case all_SD_ready :
                        /* all SDs must be in the ready state  */
                        one_not_ready = (SD_state[i] != ready) ;

                  case all_SD_copy_done :
                        /* all SDs must be in the normal state */
                        one_not_ready = (SD_state[i] != copy_done) ;
            }
      }
while ((one_not_ready) && ~(timer_expiration))
      if ~(one_not_ready) unset_timer ;
}
```

The use of the various timers is as specified in Chapter 4, excepting that separate timers *SM_timer3*, *SM_timer6*, *SD_timer3* and *SM_timer6* are used, and that expiry of *SD_timer(3)* on SD failure during bank to bank copying can be handled, as discussed in [Coghlan et al 92a].

6.6.3 Read and Write Commands

For FASST, a processor is not allowed to issue read and write commands to stable devices during the bank to bank copy. Although not strictly necessary, this constraint is retained for the SD. In the *normal* state of the SD, most of the work is concerned with dependency management. A dependency i → j is noted for :

for the SMs :

```
(M[i] & (1<<j)) == 1)
```

for the SDs :

```
(M[i, j] == 1)
```

Unlike the SM, the SD records dependencies at **page** granularity, where the conversion between a cell address and a **page** number is done by the *SD_atb* function, implemented in the hardware of the SD protection logic.

6.6.3.1 Read

A read to cell c will compute the target page b, record any dependency in the matrix *M* and will deliver the current value of the cell. In FASST the processor identifier is a 4bit field called *user*. To accomodate the SD the `read` command must be renamed `SM_read` and a new `SD_read` command defined :

```
user & read_op  <=> reader
user & write_op <=> writer
```

Thus :

```
SM_read (address, reader)
{
       block = atb (address) ;
       owner = Vector[block].owner ;
       if (owner != NIL)
             /* recording a dependency */
             M[reader] |= (1<<owner) ;
       return (bank1[address]) ;
}

SD_read (address, reader)
```

```
{
       block = SD_atb (address) ;
       owner = SD_vector[page].LM ;
       if (SD_vector.status == clean)
             /* recording a reference  */
             SD_vector.status = referenced ;
       elseif (SD_vector.status == modified)
             /* recording a dependency */
             M[reader, owner] = 1 ;
       return (bank1[address]) ;
}
```

6.6.3.2 Write

A write to cell c will compute the target page *b*, record any dependency in the matrix *M*, update *LM*, and then update the current value of the cell. Again, to accomodate the SD the `write` command must be renamed `SM_write` and a new `SD_write` command defined :

```
SM_write (address, writer, value)
{
       block = atb (address) ;
       owner = Vector[block].owner ;
       if (owner == NIL)
             /* first time this block is modified since the last commit */
             *update_ptr++ = block
       else
             /* recording a dependency */
             M[owner] |= (1<<writer) ;
       Vector[block].owner = writer ;
       Bank1[address] = value ;
}
```

```
SD_write (address, writer, value)
{
       block = SD_atb (address) ;
       owner = SD_vector[page].LM ;
       if ((SD_vector.status == clean) ||
             (SD_vector.status == referenced))
             /* first time this block is modified since the last commit */
             SD_vector.status = modified ;
       elseif (SD_vector.status == modified)
             /* recording a dependency */
             M[owner, writer] = 1 ;
             M[writer, owner] = 1 ;
       SD_vector[page].LM = writer ;
       Bank1[address] = value ;
}
```

6.6.4 Behaviour of the processor initiating a commit

6.6.4.1 Body of the initiator

The initator algorithm only needs to be adapted to accomodate the different numbers of SMs and SDs, and their different timers. As in Chapter 4, the initiator is assumed to be a member of the commit group.

```
Initiate_Commit ()
{
       /* ensure there is only one initiator         */
       /* i.e. only one commit in progress at a time */
       Obtain_Commit_Lock () ;
             :
             :
       /* Copy dependency group to stable devices    */
       Broadcast (group, SMs) ;
       Broadcast (group, SDs) ;
/*
What is important here is that all stable devices commit or none of them
does. roblems may arise if the initiator fails while he is requesting the
stable devices to commit. Some stable devices may commit while others
may rollback leading to an inconsistent state.
*/

       /* wait until all stable devices are ready to commit */
       Wait (all_SM_ready, SM_nb, SM_timer6) &&
       Wait (all_SD_ready, SD_nb, SD_timer6) ;

       Sync[my_pid] = recovery ;

       /* commit stable devices */
       Inform_sd (commit) ;

       /* wait until all stable devices have finished bank to bank copy */
       Wait (all_SM_copy_done, SM_nb, SM_timer3) &&
       Wait (all_SD_copy_done, SD_nb, SD_timer3) ;

       Inform_sd (normal) ;
             :
             :
       /* end of commit from initiator point of view */
       Sync[my_pid] = normal ;
       Release_Commit_Lock () ;
}
```

Note that the `inform_sd` function must accomodate the different SM and SD command and status register mappings.

6.6.4.2 Group computation

When P_i commits then all its decendents according to the dependency relation must commit. The commit group is obtained by computing the transitive closure of the dependency matrix. If P_j commits then every *k* which verifies the following equality has to commit too and recursively:

for the SMs :

```
(M[j] & (1<<k)) == 1)
```

for the SDs :

```
(M[j, k] == 1)
```

Symmetrically, when P_i rolls back then all its ascendents according to the dependency relation must rollback. The rollback group is obtained by computing the transitive closure of the inverse of the dependency matrix. If P_j rolls back then every *k* which verifies the following equality has to rollback too and recursively:

for the SMs :

```
(M[k] & (1<<j)) == 1)
```

for the SDs :

```
(M[k, j] == 1)
```

The `Compute_Group` primitive forms the global dependency matrix from the local matrix (located in SMs and SDs) and then computes the dependency group related to the processor *p* given as a parameter. If the *type* parameter equals *commit* this primitive computes the commit dependency group. If the *type* parameter equals *rollback* this primitive computes the rollback dependency group. This function is only affected where the dependency matrices are being read from the stable devices:

```
Compute_Group (type, p, group)
{
       /* computation of the dependency group from the dependency matrix */
       /* -- read all matrices from stable devices      */
       /* -- build the global matrix                    */
       /* -- compute dependency group                   */

       /* initialization of M with no dependency        */
       for (i=0; i<sd_nb; i++)
             M[i] = (1<<i) ;

       /* reading all matrices from stable devices      */
       /* and building the global matrix                */
       for (i=0; i<SM_nb; i++)
             for (j=0; j<p_nb; j++)
                   M[j] |= Mi[j]
       for (i=0; i<SD_nb; i++)
             for (j=0; j<p_nb; j++)
                   for (k=0; k<p_nb; k++)
                         M[j] |= (Mi[j, k]<<1)

       /* compute dependency group */
       group = (1<<p) ;
       tempo_group = new = group ;
       group_computed = false ;
       do
       {
             :
             :
       }
       while (group_computed == false)
}
```

6.6.5 Behaviour of other processors

When a commit or rollback takes place the initiator sends a *commit_interrupt* or *rollback_interrupt* to all the other processors. The algorithm that describes the behaviour of the processors other than the initiator is not affected by the presence of SDs.

6.6.6 Stable device behaviour during recovery operations

Recovery operations are started in the SMs by the initiator write to the group register. This behaviour must be modified to accomodate the different numbers of SMs and SDs, their different dependency matrices, their different status registers and their different timings.

```
main ()
{
       /*
       For the SM, recovery operations start with a write to the group
       register by the initiator processor.
       */
       Wait (group != NIL) ;

       /*
       For the SD recovery operation starts with a write to the event
       register by the initiator processor.
       */
       Wait (event != 0) ;

       /* update dependency matrices in order to break dependencies */
       for (i=0; i<SM_nb; i++)
             for (k=0; k<p_nb; k++)
             {
                   if (group & (k<<1)) Mi[k] |= (1<<k) ;
             }
       SM_state = ready ;
```

```
        for (i=0; i<SD_nb; i++)
              for (k=0; k<p_nb; k++)
              {
                    if (group & (k<<1)) Mi[k, k] = 1 ;
              }
        SD_state = ready ;

        /*
        The initiator processor can observe that the SM bank to bank copy is
        progressing by the following mechanism. It reads the bottom and top
        pointers of the update list and by their difference can compute a
        value for SM_timer(3). A very efficient method is to check
        if the working pointer is growing. */
        Wait ((SM_state == commit) || (SM_state == rollback) ;
        SM_Phase2 (SM_state)
        SM_state = copy_done ;
        Wait (SM_state == normal) ;

        /*
        The progress of the SD bank to bank copy can be observed by waiting
        for the SD interrupt status register event flag to be cleared. The
        value of SD_timer(3) can be computed from the number of pages
        modified.
        */
        Wait ((SD_state == commit) || (SD_state == rollback) ;
        SD_Phase2 (SD_state)
        SD_state = copy_done ;
        Wait (SD_state == normal) ;
}
```

The `SM_Phase2` procedure for the SM consists of the bank to bank copy of blocks whose last writer belongs to the dependency group. For the SD, `SD_Phase2` consists of the bank to bank copy of pages whose last modifier *LM* belongs to the dependency group. If a commit operation takes place the copy is done from *bank1* to *bank2*. If a rollback operation takes place the copy is done from *bank2* to *bank1*. The following algorithms are abstracted from Section 4.2.6.5 :

```
SM_Phase2 ()
{
        Gap_location = update ;
        i = update ;
        while (i<update_ptr)
        {
              block = *i ;
              owner = Vector[block].owner ;
              if (owner & Group)
              {
                    switch on type
                    {
                          case commit :
                                Bank2[block] = Bank1[block] ; break ;
                          case rollback :
                                Bank1[block] = Bank2[block] ;
                    }
                    Vector[block].owner = NIL ;
              }
              else
              {
                    *Gap_location = *i ;
                    Gap_location++ ;
              }
              i++ ;
        }
}

SD_Phase2 ()
{
        i = 0 ;
        while (i<SD_mem_size/2)
        {
                owner = SD_vector[i].LM ;
                if (owner & Group)
                {
                        switch on type
                        {
                                case commit :
                                        SD_bank2[i] = SD_bank1[i] ; break ;
                                case rollback :
                                        SD_bank1[i] = SD_bank2[i] ;
                        }
                        SD_vector[i].state = clean ;
                }
                i++ ;
        }
}
```

Note that the algorithm for SD_Phase2 can be optimized by use of the *marked* state to allow the processes to restart execution while the bank to bank copy proceeds, with either rescheduling or priority copy of a bank if it is updated during the bank to bank copy process.

6.6.7 Atomic operations

Chapter 4 proposes a mechanism to allow for atomic critical section implementation, where no other processors are executing. The mechanism is composed from standard low level functions. The action of SDs is buried within the low level functions involved so that at the higher level the mechanism is unaffected by the presence of SDs.

6.6.8 Rollback due to a processor failure

We assume processor P_i detects the failure of processor P_k. P_i initiates a rollback of the set of processors which are dependent on P_k. Two situations must be considered.

Firstly, what if a recovery operation is in progress. The mechanisms to handle all permutations of this event are not defined in Chapter 4, but since most actions invoked are likely to be a compendium of standard low level functions, it is expected that the SD will integrate without difficulty.

Secondly, what if no recovery operation is in progress. The algorithms for two approaches to this are outlined below.

6.6.8.1 Rollback of the dependency group

This algorithm minimizes the number of processors that are *stopped* at the expense of algorithmic complexity. The algorithm only needs to be adapted to accomodate the different numbers of SMs and SDs, and their different timers.

```
Initiate_Rollback (k)
{
        /* first, check if there is a commit in progress */
                :
                :
                /* copy dependency group to stable devices */
                Broadcast (group, SMs) ;
                Broadcast (group, SDs) ;

                /* wait until all stable devices are ready to rollback */
                Wait (all_SM_ready, SM_nb, SM_timer6) &&
                Wait (all_SD_ready, SD_nb, SDtimer6) ;

                Sync[my_pid] = recovery ;

                /* rollback stable devices */
                Inform_sd (rollback) ;
```

```
             /* wait until all stable devices have finished    */
             /* the bank to bank copy                          */
             Wait (all_SM_copy_done, SM_nb, SM_timer3) &&
             Wait (all_SD_copy_done, SD_nb, SD_timer3) ;

             Inform_sd (normal) ;
                   :
                   :
             /* end of rollback from initiator point of view   */
             Sync[my_pid] = normal ;
             Release_Commit_Lock () ;
       else
             {
                   /* a commit is in progress !!! */
             }
}
```

6.6.8.2 Rollback of all processors

This algorithm minimizes the algorithmic complexity at the expense of all processors being *stopped*. Again, the algorithm only needs to be adapted to accomodate the different numbers of SMs and SDs, and their different timers.

```
Initiate_Rollback (k)
{
       /* first, check if there is a commit in progress */
             :
             :
             /* copy dependency group to stable devices */
             Broadcast (group, SMs) ;
             Broadcast (group, SDs) ;

             /* wait until all stable devices are ready to rollback */
             Wait (all_SM_ready, SM_nb, SM_timer6) &&
             Wait (all_SD_ready, SD_nb, SDtimer6) ;

             Sync[my_pid] = recovery ;

             /* rollback stable devices */
             Inform_sd (rollback) ;

             /* wait until all stable devices have finished    */
             /* the bank to bank copy                          */
             Wait (all_SM_copy_done, SM_nb, SM_timer3) &&
             Wait (all_SD_copy_done, SD_nb, SD_timer3) ;

             Inform_sd (normal) ;
                   :
                   :
             /* end of rollback from initiator point of view   */
             Sync[my_pid] = normal ;
             Release_Commit_Lock () ;
       else
             {
                   /* a commit is in progress !!! */
             }
}
```

6.6.9 Various primitives used in the protocol description

6.6.9.1 Updating and consulting synchronization registers

The `Inform_p` primitive is used by the initiator to update the synchronization registers of a set of processors group with the value *state*, and as such is not affected by the presence of SDs.

The `Inform_sd` primitive is used to update stable device synchronization variables. This is implemented via commands in the prototype, and must be altered :

```
Inform_sd (state)
{
       for (i=0; i<SM_nb; i++)
             SM_state[i] = state ;
       for (i=0; i<SD_nb; i++)
             SD_state[i] = state ;
}
```

6.6.9.2 Locking

Only one commit or rollback is allowed at the same time. This is ensured by using a global lock. This operation is not affected by the presence of SDs.

6.7 Influence on performance[2]

Secondary storage records are buffered in a Stable Memory. This is a fast operation, since it only involves the system bus. The RAID Controller is designed to then transfer the records to/from the buffer with minimal impact on the host processors. This process is illustrated in Figure 6.19. How does this affect performance ?

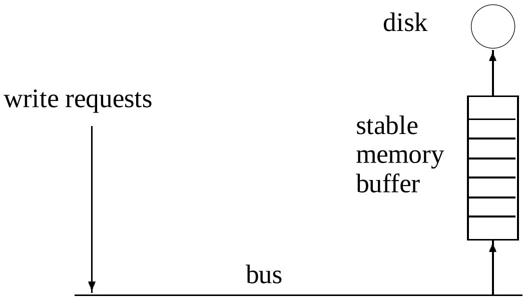


Figure 6.19: Buffered disk requests

To evaluate the performance of the Stable Disk, we model the system as a network of two nodes in tandem, such as the one in Figure 6.20. The first node represents the bus. Jobs (output requests generated by the processors) arrive there in a Poisson stream with rate λ, and join an unbounded queue. After completing service at node 1 (exponentially distributed with parameter μ), they proceed to node 2, which represents the Stable Disk. At node 2 there is a finite buffer with room for a maximum of N jobs (including the one in service). If, at the start of a bus service, the buffer is full, the bus waits until the completion of the current service at node 2 (exponentially distributed with parameter ξ). In this last case, server 1 is said to be 'blocked'. Transfers from node 1 to node 2 are instantaneous. Since bus operations are usually much faster than writing to a disk, we typically have $\mu >> \xi$.

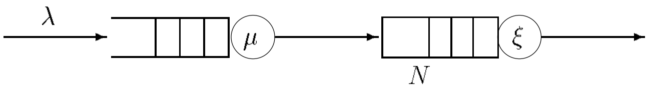


Figure 6.20: A two-node tandem network model

[2] This section contributed by Isi Mitrani, Department of Computing Science, University of Newcastle, Newcastle upon Tyne NE1 7RU, U.K.

The above type of blocking is referred to as ‘communication blocking’, to distinguish it from ‘manufacturing blocking’, where node 1 checks the buffer *after* completing a service.

In this system, the unbounded queue at node 1 operates in a finite-state environment defined by node 2. We say that the environment, X_t, is in state i if there are i jobs at node 2 $(i = 0, 1, \ldots, N)$. Of course if $i = N$ and the queue at node 1 is not empty, then server 1 is blocked.

The pair $U = \{(X_t, Y_t)\ ;\ t \geq 0\}$, where Y_t is the number of jobs at node 1, is an example of a ‘Quasi-Birth-and-Death’ process. To obtain performance measures of interest, it is necessary to determine the joint steady-state distribution of X and Y :

$$p_{i,j} = \lim_{t\to\infty} P(X_t = i\,,\, Y_t = j)\ \ ;\ \ i = 0, 1, \ldots, N\ \ ;\ \ j = 0, 1, \ldots\ . \tag{6.6}$$

These probabilities can be computed by the so-called *spectral expansion* method. We give a brief outline of the analysis here. For more details, the reader is directed to [Mitrani 95] and [Mitrani et al 95], and the references therein.

If the system is in state (i, j), with $i > 0$, it can move to state $(i-1, j)$ with rate ξ. In other words, the transition rate matrix, A, which controls the changes of the environment without altering the unbounded queue, is given by

$$A = \begin{bmatrix} 0 & & & \\ \xi & 0 & & \\ & \ddots & \ddots & \\ & & \xi & 0 \end{bmatrix} .$$

Since the arrival rate into node 1 does not depend on either i or j, the transition rate from state (i, j) to state $(i, j+1)$ is λ, for all $i, j \geq 0$. That is, the matrix of transitions which increase the queue size is $B = \lambda I$, where I is the identity matrix of order $N + 1$.

The departures from node 1 are always accompanied by environmental changes: from state (i, j) the process moves to state $(i+1, j-1)$ with rate μ for $j > 0$ and $i < N$. In state (N, j), server 1 is blocked and there can be no departure from node 1. The matrix, C, corresponding to these transitions is equal to

$$C = \begin{bmatrix} 0 & \mu & & & \\ & 0 & \mu & & \\ & & \ddots & \ddots & \\ & & & 0 & \mu \\ & & & & 0 \end{bmatrix} .$$

Define the row vectors of probabilities corresponding to states with j jobs at node 1:

$$\mathbf{v}_j = (p_{0,j}, p_{1,j}, \ldots, p_{N,j})\ \ ;\ \ j = 0, 1, \ldots\ . \tag{6.7}$$

Also, let D^A, D^B and D^C be the diagonal matrices whose i th diagonal element is equal to the i th row sum of A, B and C, respectively. Then the steady-state balance equations for $j > 0$, can be written as:

$$\mathbf{v}_j[D^A + D^B + D^C] = \mathbf{v}_{j-1}B + \mathbf{v}_j A + \mathbf{v}_{j+1}C\ \ ,\ \ j = 1, 2, \ldots\ . \tag{6.8}$$

For $j = 0$, the equation becomes

$$\mathbf{v}_0[D^A + D^B] = \mathbf{v}_0 A + \mathbf{v}_1 C\ \ ,\ \ j = 1, 2, \ldots\ . \tag{6.9}$$

In addition, all probabilities must sum up to 1:

$$\sum_{j=0}^{\infty} \mathbf{v}_j \mathbf{e} = 1\ , \tag{6.10}$$

where $\mathbf{e}$ is a column vector with $N + 1$ elements, all of which are equal to 1.

To find the general solution of the vector difference equation (6.8), form the matrix polynomial

$$Q(x) = Q_0 + Q_1 x + Q_2 x^2\ , \tag{6.11}$$

where $Q_0 = B$, $Q_1 = A - D^A - D^B - D^C$ and $Q_2 = C$. Denote by x_k and $\mathbf{u}_k$ the 'generalized eigenvalues', and corresponding 'generalized left eigenvectors', of $Q(x)$. In other words, these are quantities which satisfy

$$det[Q(x_k)] = 0 \ ,$$

$$\mathbf{u}_k Q(x_k) = \mathbf{0} \ \ ; \ \ k = 1, 2, \ldots, d \ , \tag{6.12}$$

where $det[Q(x)]$ is the determinant of $Q(x)$ and d is its degree. In our case all eigenvalues are real, positive and simple. Moreover, $N+1$ of them are in the interval (0,1). Let the numbering be such that those are the first $N+1$ eigenvalues, $x_1, x_2, \ldots, x_{N+1}$. Then the solution of (6.8) can be expressed as a linear combination,

$$\mathbf{v}_j = \sum_{k=1}^{N+1} \alpha_k \mathbf{u}_k x_k^j \ \ ; \ \ j = 0, 1, \ldots \ , \tag{6.13}$$

where α_k ($k = 1, 2, \ldots, N+1$), are some constants. The latter are determined from the balance equations (6.9) (N of them are linearly independent), and the normalizing equation (6.10).

Having determined the coefficients in the expansion (6.13), it is easy to compute performance measures. The steady-state probability that the environment is in state i (i.e., there are i write requests in the buffer), is given by

$$p_{i,\cdot} = \sum_{k=1}^{N+1} \alpha_k u_{k,i} \frac{1}{1 - x_k} \ , \tag{6.14}$$

where $u_{k,i}$ is the i th element of $\mathbf{u}_k$.

The conditional average number of jobs in the system, L_i, given that the environment is in state i, is obtained from

$$L_i = \frac{1}{p_{i,\cdot}} \sum_{k=1}^{N+1} \alpha_k u_{k,i} \frac{x_k}{(1 - x_k)^2} \ . \tag{6.15}$$

The overall average number of jobs in the system, L, is equal to

$$L = \sum_{i=0}^{N} p_{i,\cdot} L_i \ . \tag{6.16}$$

Finally, the average response time of a write request is given by $W = L/\lambda$.

As an illustration of the results that can be obtained from this model, Figure 6.21 shows the minimum buffer size required to achieve a given average response time. That size is plotted against the arrival rate, for different Stable Disk service times. This confirms what one might have expected.

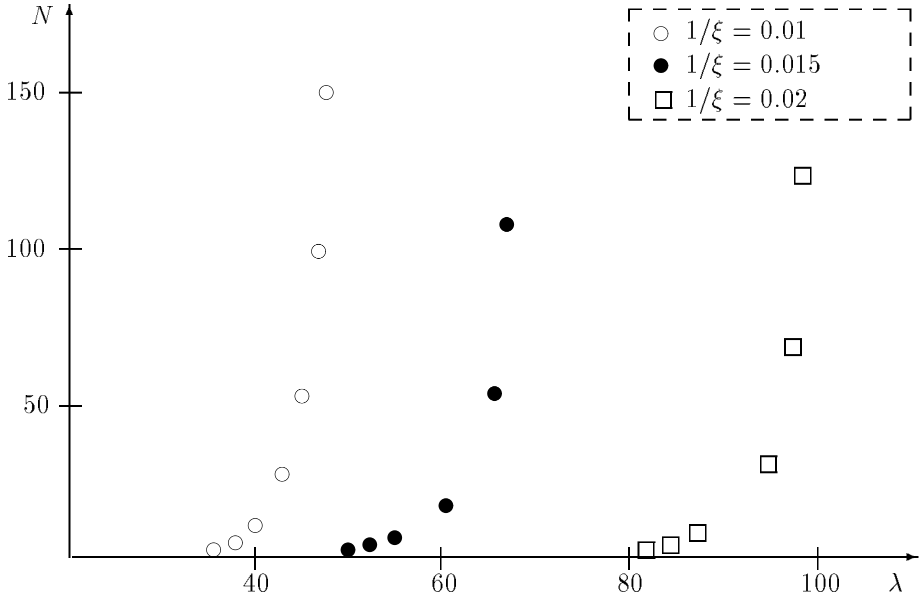


Figure 6.21: Value of N such that $W \leq 0.05$ sec. $1/\mu = 0.0001$

Chapter 7

System Software

7.1 System software principles[1]

Two basic structuring principles are applied in the system software of modern computers. First, the system may be organized as a hierarchy of layers, each one constructed upon the one below it. Second, system services may be provided as a set of communicating processes. To request a service, a process (referred to as a *client*) sends a request to a *server* process which then does the work and sends back the answer. Figure 7.1 illustrates these principles.

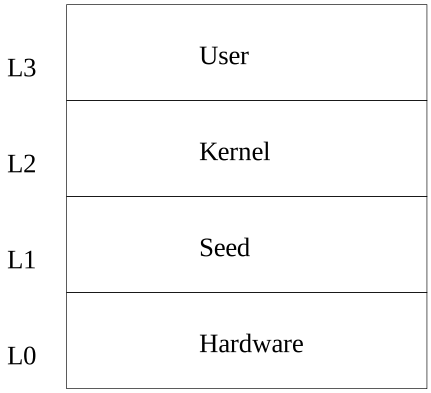


Figure 7.1: Structure of the system software

Level L_0 denotes the hardware, including both the SM and the SD (for brevity, hereafter when we refer to the SM, the equivalent functionality of the SD is also implied). Level L_1, referred to as the $seed$, deals with the basic process management. Above the seed, the operating system kernel services (L_2) can then be structured as a set of communicating processes. Finally, level L_3 deals with users which may request the services of the kernel from their application programs. In the following, we discuss the design principles of these layers with particular emphasis on recoverability but first consider the recovery provision for the basic model of computation as introduced in Section 1.4.

7.1.1 Providing recoverability for the basic model of computation

Recall that in the basic model of computation, a process may access a local state (process registers) and a shared state which we assume to be represented in the SM. Let us first examine how recoverability can be provided for this model considering the simplifying case where a separate processor would be dedicated to each process.

Recoverability of a system of communicating processes responding to the basic model is simply provided by mapping a *process transaction* to a *processor transaction*. When a new process transaction is started (following an explicit request of the process or an implicit action of the recovery protocol), the local state of the process (the registers of the processor executing the process) is written into memory, the cache if any is flushed into the SM, and the current (active) processor transaction associated to the processor executing the process is committed. The processor begins a new processor transaction.

The facilities offered by the SM are almost sufficient themselves for implementing correctly the model of computation defined previously and are very adequate for tolerance to hardware faults (processor failures). When a processor failure occurs, the current transaction is rolled back and the process can safely restart its computation on another processor after having loaded the process local state from the SM into the registers of the new processor allocated to the process. The set of processors forced to roll back their current transaction due to the recovery protocol must also load their registers with the values safely stored in the SM on the last transaction commitment before pursuing forward activity. The other processors are not affected.

In summary, the fault tolerance mechanisms of the SM are (almost) sufficient to cope fully with processor failures so as to make these failures *transparent* to the processes obeying the basic model of computation.

[1] This section contributed by Michel Banâtre, Maurice Jégado, Philippe Joubert and Christine Morin, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France

Our conjecture is that this is also the case for any abstract model of computation that can be mapped onto this basic model. A rough argument of this (conjectured) property, based on abstract data type theory, might be as follows. Consider a program $\mathcal{P}$ responding to an abstract model of computation, starting in an initial abstract state s and terminating in a final abstract state f denoted $\mathcal{P}(s) = f$. Assume that $\mathcal{P}$ is mapped to a program $\mathcal{P}_0$ responding to the basic model of computation such that $\mathcal{P}_0(s_0) = f_0$; s_0 and f_0 denoting respectively the initial and final concrete states of $\mathcal{P}_0$. Let abs be the abstraction function from the concrete domain onto the abstract domain. Assume that a correct mapping satisfies the following predicate: $((\mathcal{P}_0(s_0) = f_0 \wedge \mathcal{P}(s) = f \wedge abs(s_0) = s) \Rightarrow abs(f_0) = f)$. Should a component process of $\mathcal{P}_0$ be rolled back due to a processor failure, the recovery protocol constructs a program $\mathcal{P}_0'$ equivalent to $\mathcal{P}_0$ (see Section 1.4) leading to the same final concrete state f_0. It is then easy to see that the abstract program $\mathcal{P}$ is not influenced by the recovery actions since the abstract final state of $\mathcal{P}$ will be the same. Clearly, a more formal proof of this property would be desirable.

In the absence of a formal proof, let us assume that it is so. We may extend recoverability to the more realistic case where a multiprocessor may support the execution of an arbitrary number of processes competing for a limited number of available processors by introducing the concept of a *seed*.

7.1.2 The seed

The role of the seed is to deal with the basic process management and to provide higher layers with a useful model of communicating sequential processes. In addition to the basic model of computation introduced in Section 1.4, we require that the seed offers abstract synchronisation and communication primitives, and allows for the dynamic creation and deletion of processes. The seed is needed to hide machine-dependent features (e.g. interrupt handling) and thereby provide a machine independent interface that facilitates the portability of kernel services. The bulk of memory management is not considered as being part of the seed but as a kernel service running at a higher layer. This is discussed below in Section 7.1.4. Conceptually, the seed layer has many similarities with microkernels that have been discussed in the operating system literature, such as Chorus [Rozier et al 88] or Mach [Baron et al 88].

7.1.2.1 Virtualization of the processor resource

Since dedicating a physical processor to each process is not realistic in a classical multiprocessor architecture where the number of available processors is limited, our first step is to virtualize this resource. A *virtual processor* is allocated to each process.

Time sharing is a well known technique to virtualize the processor resource. On a monoprocessor, the physical processor is allocated to each virtual processor for a time quantum. Virtual processors may be managed by a short term *scheduler* according to a round robin discipline. This same technique easily extends to a multiprocessor. We assume, as it is generally the case, that a virtual processor may be mapped to different physical processors during its activity.

The most straightforward implementation of process recoverability in this context is to conceal the scheduling activity and provide the notion of *virtual processor transaction*. A single active virtual transaction is associated to each virtual processor; a process transaction is mapped to a virtual processor transaction. In the following, the word transaction (alone) stands for virtual processor transaction.

Consider now the scheduler design in greater detail. We assume that a zone of the SM is allocated to each process for *stacking* private data. We also assume that once a process has consumed its time quantum on a processor, some clock device sends an interrupt to the processor. This has the effect to copy the local state of the process on top of its private stack, and triggers the execution of the scheduler. Symmetrically, returning from the scheduler has the effect to pop the local state at the top of the current stack into the processor's registers. We assume that a *context* is allocated at a fixed address in the SM for containing the process local state together with a link field used for list management. An array $active$ such that $active[K]$ refers to the context of the active process on processor K is maintained by the scheduler. A single list $ready$ protected by a global lock chains the contexts of the processes that are not active. When the scheduler is entered by processor K because the active process on processor K has consumed its time slice, the active process is descheduled and chained at the tail of the $ready$ list to the benefit of the head process of the ready list which is removed from the list and made active. In other words, the set of processes contained in the system are the processes pointed to by the $active$ array plus those chained in the $ready$ list. This set is maintained as an invariant by the scheduler (if we ignore the dynamic creation/deletion of processes).

Without further constraints on scheduling, a processor transaction will embed activities belonging to distinct transactions, and a transaction will be mapped to several processor transactions. To overcome this, let us commit a transaction each time a virtual processor is allocated a physical processor by the scheduler. The basic steps performed by the scheduler entered on processor K by process p are then as described by Figure 7.2.

```
(1)        Save the registers' values residing on top of the stack of $p$
                                                into the context of $p$.
(2)        Commit the current processor transaction.
             Now $active[K] = p$ and $ready = q + X$
(3)        Schedule a new process $q$.
             Now $active[K] = q$ and $ready = X + p$
(4)        Begin a processor transaction.
(5)        Install the context of $q$ on top of the stack of $q$.
(6)        Return from the scheduler
```

Figure 7.2: The short-term scheduler: specifications

Consider the treatment of a failure of processor K, assuming for the moment that this event cannot occur while performing the scheduling sequence above. Let P_i denote a processor member of the set of the rolled back processors due to the failure of processor K. This set is automatically determined by the recovery protocol. Basically, recovering from a processor failure boils down to the following actions:

(1) The context pointed to by $active[P_i]$ must be loaded into the registers of processor P_i. Alternatively, this context could be inserted into the ready list and a new process dispatched on P_i. This action can be performed by processor P_i itself.

(2) The process whose context is pointed to by $active[K]$ must be inserted into the ready list. This action must be performed by a processor *elected* out of the remaining live processors (ignoring the detail of such an algorithm). The process will then be dispatched on another live processor by the scheduler.

One can see that the failure of a processor does not affect all the processors of the machine, but only those that are dependent on the failing one.

Consider now the case where processor K may fail while performing the scheduling sequence. Logically, the data structures maintained by the scheduler itself are not part of the shared state of a process given our basic model of computation. How should recoverability be provided for those objects ? A first possibility is to exclude those data structures from the recovery protocol, recoverability being then achieved by explicit *forward recovery* [Randell et al 78]. A second possibility is to attempt to include those structures within the recovery protocol. We illustrate the second possibility in the algorithm of Figure 7.3.

```
(1)        pop(stack[active[K]], context[active[K]])
(2)        NewProcessorTransaction
(3.1)test: test_and_set (val, GlobalLock)
(3.2)      if val <> 0 then goto test fi
(3.3)      PutTail(ready, active[K])
(3.4)      RemoveHead (ready, active[K])
(3.5)      GlobalLock := 0
(4)        NewProcessorTransaction
(5)        push(stack[active[K]], context[active[K]])
(6)        return
```

Figure 7.3: The short-term scheduler implementation

Fortunately, for Figure 7.3, if we can guarantee that the section (3.1-3.5) is an atomic action, then failure while performing this scheduling sequence is equivalent to the previous situation where the failure occurs outside of the scheduler. Bracketing this sequence within appropriate primitives (*NewProcessorTransaction*) would not normally be sufficient to ensure the atomicity property, since implicit commitments due to the recovery protocol might occur while performing the action. However, in this particular case, within the seed itself where the programmer has a control over the implicit commitments performed by the recovery protocol, a cheap way to ensure the atomicity of the above action is to defer the treatment of any commit request until the next *NewProcessorTransaction* primitive is encountered. Naturally, the defer period should be short (which is the case here), since other processors could be blocked waiting for the end of the commit protocol.

We can also observe that a commit request cannot happen while performing section (3.1-3.5), since it is tantamount to a processor critical section. By definition, this section can only be entered by a single processor at a time, and the processor within the section cannot be a potential recovery initiator of another processor willing to commit.

As a final remark, notice that the *NewProcessorTransaction* primitive of Figure 7.3 does not need to trigger a register flush, but only a cache flush (if any), followed by a call to the do_commit command of the SM.

7.1.2.2 Basic synchronisation primitives

Let us assume that processes can exchange $messages$ through shared memory objects called $ports$. To simplify the presentation, we assume that a port may retain the memory of an arbitrary number of messages, and that a message may contain a variable size collection of data, although a particular implementation may restrict these hypotheses. Sending a message m to a port p (i.e. *send (p, m)*) is assumed to be an asynchronous operation while receiving a message from a port (i.e *receive (p, m)*) is assumed to be blocking if the port is empty.

Recoverability for this model of computation can be provided in almost the same manner as explained for the basic model. What is required is that a dependency must be recorded between a process sending a message and the process receiving it. But, as ports are memory objects, the SM hardware will automatically do this.

It should be noted that the default dependency tracking policy provided by the hardware may lead to more dependencies than strictly required by the model of computation due to the access to the concrete data representing the abstract model of computation. For performance reasons, it may be better to have an explicit control mechanism to escape the default policy in favour of an explicit programmed dependency tracking policy.

We will not go into implementation details of the basic synchronisation primitives here, but just mention that the scheduler algorithm discussed above is a sound basis upon which to build them.

7.1.3 Standard vs non-standard processes

So far, we have been considering that a process may access a local state (process registers), a shared state in the SM, and ports which are assumed to be represented in the SM. We say that these objects are *implicitly* recoverable via the recovery protocol. In a realistic architecture, there are clearly other objects to be controlled by an operating system for which recoverability might not be implicitly provided, particularly I/O devices. A process which accesses a local state and implicitly recoverable objects will be referred to as a *standard* process while a process which accesses objects that are not implicitly recoverable will be referred to as a *non-standard* process [Banâtre et al 92a]. Given this distinction, we may expect the kernel services to be programmed as a set of standard and non-standard communicating processes. User processes should be standard in the sense that recoverability should be transparent to them.

Programming a non-standard server process will depend on the type of unrecoverable objects the process is dealing with. However, it is interesting to propose some programming guidelines. These are discussed in the following.

7.1.3.1 General principles for non-standard servers

If each service of a server is programmed as a *restartable* operation O, servicing a request despite a processor failure can be obtained in the following way [Lampson 81a]:

- **(1)** Save the server's context in stable storage,
- **(2)** Perform O, and
- **(3)** Erase the server's context from stable storage.

If a processor failure occurs while performing O, the process will resume after (1) and will perform O again, the resulting execution sequence is equivalent to a single execution of O by definition.

Given our model of computation, we may embed the server's operation O within two *NewProcessTransaction* primitives so as to ensure service despite failures. There are two aspects of this to consider. First, recall from

Section 1.4 that such a region may be dynamically broken into multiple transactions due to the implicit commitments performed by the recovery protocol. Consequently, the restartable property (if any) of a service does not lead immediately to a solution, in contrast to above. Second, O will in general be a compound action made of both recoverable and non-recoverable actions.

The first situation is particularly embarassing. We may think of providing explicit *kernel transactions* (or seed transactions) fully under the programmer's control, as in [Banâtre et al 91c], leaving the programmer to cope only with providing recoverability to the non-recoverable objects used within a kernel transaction. Providing kernel transactions is a very large task. Instead, here we discuss using only the properties of the services to be programmed, together with some additional provisions, to program a limited number of non-standard servers.

The first provision is a limited amount of non-recoverable memory that may be used to record the state of objects for which explicit recovery is needed. We define a non-recoverable memory cell as one where its contents are not restored should its current process transaction be rolled back.

Secondly, in many cases, once its current transaction is rolled back, a non-standard process will wish to perform exceptional work before proceeding. Let us assume that the seed provides an exceptional mechanism *RollBackAt(address)* for that purpose, where the flow of control of the process will be resumed at *address* should the current process transaction be rolled back. Such a mechanism might be triggered in different ways by the calling process. First, *RollBackAt* might be provided as an explicit seed primitive. Second, the rollback *address* might be provided as an explicit exceptional *continuation* [Livercy 78] argument to each seed primitive. Third, if the programming language offers a mechanism for handling exceptions, it is appropriate to map the rollback of a process transaction onto an exception which may then be handled according to the rules defined by the language.

Finally, a non-standard server might wish to explicitly commit its current transaction. Let us assume the seed offers a primitive, *NewProcessTransaction(p)*, for that purpose, where p denotes a process. Basically, a call to this primitive will execute a code sequence similar to the scheduler sequence depicted in Figure 7.3.

7.1.3.2 Communication issues

A client process requests the service of a server by sending its request on the server port. Conversely, the server replies to the request by sending its reply on the client port. As far as communications are concerned, it is clear that communications between standard processes (inside the standard domain) themselves do not raise difficulties. In contrast, communications with non-standard processes need to obey a particular protocol.

In order to facilitate the provision of fault tolerance measures within a non-standard server, the server might require that the client $commits$ its request before further action by the server. In other words, the client's request is an *intention* [Lampson 81a] that has to be performed by the server. The underlying reason is that in general it will be easier for a non-standard server to restart the processing of a request than to be possibly obliged to cancel the processing of a request (an $orphan$ execution) retracted by a rolled-back client. In the general case, a client's call will give rise to nested calls, which to be cancelled would require recursive cancellation of all orphan executions raised by the call. Communications with non-standard processes is likely to occur very often and therefore an efficient implementation is necessary. This dictates that the protocol must be implemented at the bottom layer (the seed) so as to use the hardware facilities in the most efficient way.

7.1.3.3 Guidelines for non-standard servers

In summary, the following guidelines for programming non-standard servers can be proposed:

(a) A non-standard server may make use of non-recoverable memory in order to record the recovery data of some objects.

(b) A non-standard server may provide a handler for dealing with a rollback of its current process transaction (triggered by the recovery protocol due to a processor failure).

(c) A non-standard server may explicitly commit its current process transaction (recall, however, that a process does have not full control over its transactions, since the recovery protocol may itself implicitly commit its current transaction).

(d) A non-standard server may require a particular communication protocol so as to facilitate the provision of its fault tolerance measures.

7.1.4 Memory management

Memory management is a central and complex component in any operating system. Our aim, in this section, is not to describe in detail the intricacies of sophisticated virtual memory management schemes (this has been discussed elsewhere [Krakowiak 85]) but to attempt isolate the new problems that virtual memory management may raise as far as the recovery protocol is concerned. For the purpose of illustration, we introduce below the main features of a memory management scheme (a particular implementation may not correspond exactly to this example, but this framework is adequate for us to illustrate the issues discussed).

We assume that that a *shared segmented virtual space* is provided to processes. A process references a word within a segment by a tuple <*SegmentId, SegmentOffset*> where *SegmentOffset* denotes the offset from the beginning of the segment. A segment is a linear address space. For the purpose of physical memory management, each segment is paged. A *SegmentOffset* gets decomposed into a further tuple <*SegmentPageNo, PageOffset*>. We assume that swap space is available for extending the capacity of the memory. A segment page may then be resident in memory or not. In the former case, we assume that the page has a *single* copy in memory. In the latter case, the page is resident in swap space and can be brought into memory if necessary. We assume a perfect model for swap space, i.e. writing a page is assumed to take place correctly, and reading a page is assumed to return the correct value.

The model of computation provided above the memory management (residing on top of the seed) is identical to the one provided by the seed itself except that now dependencies between processes must be (logically) tracked on the $virtual$ addresses referenced by the communicating processes. Ideally, we would like the SM hardware to still provide the necessary abstraction, even though the SM is only aware of the physical accesses. It is clear that as long as a segment page is not relocated, the SM provides the necessary abstraction, since a segment page can have only a single copy in physical memory. If relocation activity only involved standard processes, we may also convince ourselves that the SM will achieve the necessary abstraction, since a dependency will be recorded between a process accessing a page p before relocation and a process accessing p after relocation, and the relocation activity itself must access both physical locations. Unfortunately, in a realistic environment paging will probably involve non-standard processes. In that case (without further constraints and assumptions), the SM cannot by itself provide the necessary abstraction. Consider, for instance, the case where swapping out and swapping in pages are performed by separate non-standard server processes. Logically, swapping in a given page is dependent on the last swapping out of the page, but without further assumptions, the SM will not record this dependency as these activities do not operate within the standard domain.

There are several ways to overcome this difficulty. At one extreme, swapping out and swapping in might be programmed so as to access explicitly common SM objects, for which the hardware will implicitly track the dependencies. At the opposite extreme, commitment might be forced when relocation is performed so as to ensure that within a group of dependent process transactions, a given virtual access cannot be mapped to distinct physical locations.

A simplifying principle might be to consider that swap space contains only committed data, since then rolling back a transaction will not affect the swap space. This principle does not appear very restrictive, for if a page is not committed, it is likely to be part of the working set of a process anyway, and therefore should reside in memory. For performance reasons, we would not wish commitment to involve any swap space operation either.

7.2 The Mach Microkernel[2]

We have noted that the seed has many similarities to existing microkernels. Development of a microkernel is not a trivial exercise. Not only is it not for the faint-hearted, existing examples represent hundreds of man-years of effort. In recent years, two microkernels have predominated : the public-domain Mach 3.0 microkernel [Accetta 86, Rashid 86b] and the proprietary Chorus microkernel [Gien 90, Rozier et al 88]. Later we will examine how the seed concepts can be introduced into one of these, Carnegie-Mellon's Mach. Let us first examine Mach in some detail. Those familiar with Mach and OSF1/mk may skip the following sections and continue reading from Section 7.6.

The history of Mach can be traced back to a research project called RIG (Rochester Intelligent Gateway) which began at the University of Rochester in 1975. The main research goal of RIG was to demonstrate that an operating system can be organized and structured in a modular way, as a collection of processes which communicate

[2] This section contributed by Henry Chung and Danny Keogan, Department of Computer Science, Trinity College Dublin

by message passing. The system was designed and built, and showed that such an operating system could be constructed.

In 1979 one of the RIG designers, Richard Rashid, left the University of Rochester and moved to Carnegie-Mellon University (CMU). Rashid continued the work on message passing operating systems. He developed a new operating system for PERQ[3] workstations, called Accent. Compared with RIG, Accent had added protection, plus transparent network operations, 32bit virtual memory, and other features. An initial version was up and running in 1981.

By 1984, Accent was used on over 150 PERQs, but clearly it was less popular than UNIX. This led Rashid to design his third operating system, which he called Mach. By making Mach compatible with UNIX, Rashid hoped to be able to use the large volume of existing software for UNIX. Mach improved a lot on Accent, with threads, a better interprocess communication mechanism, multiprocessor support, and a highly advanced virtual memory system. The first version of Mach was released for the VAX 11/784, a four-CPU multiprocessor machine, in 1986. Later, Mach was selected by the U.S. Department of Defence's Advanced Research Agency (DARPA) as part of its Strategic Computing Initiative. It was made compatible with 4.2BSD by combining Mach and 4.2BSD into a single (monolithic) kernel. Shortly thereafter the Open Software Foundation (OSF) chose Mach as the basis for their operating system.

The monolithic kernel was quite large. In 1989, CMU removed all the Berkeley UNIX code from the monolithic kernel, and put it into the user space. What remained was pure Mach, and was much smaller. This was called the Mach 3.0 microkernel, and the subsequent OSF1/mk operating system releases were based on this microkernel.

The Mach microkernel was designed using object-oriented design techniques. The microkernel can be viewed as a collection of concurrent objects which communicate with each other by message passing. There are five fundamental programming abstractions out of which more complex objects are built. Each of these abstractions is itself a Mach object:

Tasks Mach breaks the UNIX view of a process down into two distinct entities: tasks and threads. Tasks are containers which hold resources; they possess an address space, a port name space and one or more threads.

Threads Threads are points of execution. They maintain minimal machine state: a stack, a set of registers and a set of thread-specific port rights. A thread belongs to only one task and gains access to resources through belonging to the task which contains them.

Ports All interprocess communication (IPC) in Mach takes place over ports. Ports are unidirectional communication channels. Ports are location-transparent. This allows services to be distributed over different address spaces on the same machine or on different machines on a network without modifying client programs. This feature is fundamental in allowing Mach to run on uniprocessor, multiprocessor, multicomputer and networked workstation clusters. It is also provides the ability for the microkernel to move the implementation of traditional OS services such as paging or device management into user-space application programs, one of the most important characteristic of Mach. The port system also provides access control between objects. A thread gains access to a port if its parent task owns a capability known as a port right. There are three types of port rights: *send*, *receive* and *send once*. Further security is provided by the fact that port rights do not have global names; instead each task maintains a port name space. A particular port right only has meaning within the context of this name space. This prevents a task attempting to defeat the access control by sending an illicit port right to another task.

Messages Messages are typed collections of data which are sent through ports. The interface between Mach objects is defined in a language independent interface description language (IDL), which describes the format of the messages handled by the object. These IDL descriptions are compiled with the Mach Interface Generator (MiG) into a set of stub routines in a target language. A thread sends a message by calling a MiG stub routine. Messages are received by a thread through parameters passed by reference to the MiG stub in the case of a synchronous call, or via a callback for an asynchronous call. In this way, all the details of the Mach IPC system are hidden or encapsulated in a procedure call interface.

Memory Objects The microkernel provides a powerful and flexible virtual memory system. Memory objects form the basis of this system. Each range of virtual memory is backed by a memory object which is responsible for providing the microkernel with the data in that range on demand.

[3] PERQ was an early engineering workstation, with a bitmapped screen, mouse and network connection.

The microkernel uses the five fundamental abstractions described above to build an environment tailored to emulating OS personalities. The microkernel encapsulates process management, virtual memory management and device management in a hardware independent manner so that OS servers, the application programs which provide these personalities, contain little hardware dependent code. It is possible to have multiple OS servers running simultaneously.

7.3 OSF1/mk[4]

The Open Software Foundation (OSF) was founded in 1988 by a group of major computer manufacturers[5] to develop and deliver software for open systems. OSF1/mk is an operating system based on the Mach 3.0 microkernel and various servers running in user mode on top of it. Servers are available for the popular variants of UNIX, such as BSD, System 5 and HP/UX. Figure 7.4 depicts an overview of its main software components:

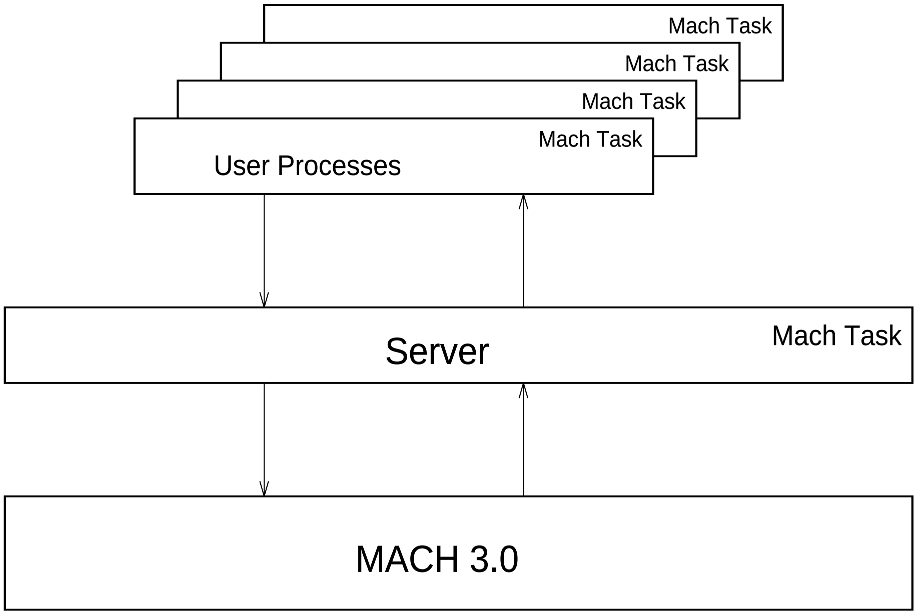


Figure 7.4: Overview of OSF1/mk software architecture

Mach 3.0 kernel The microkernel is the only machine dependent component of OSF1/mk.

UNIX Server The UNIX server performs the system calls requested by user processes using the primitives provided by Mach 3.0.

User Processes These are the applications to be executed by the computer; they use Mach primitives to communicate with the server to request the execution of system calls.

Mach 3.0 is a general purpose microkernel, so there is a lot of literature about it. [Accetta 86, Loepere 92a, Loepere 92b, Rashid 86b] are a good starting point. The details given here are appropriate to Version 4.1 of OSF1/mk.

7.3.1 BSD Server

At the time when the OSF was looking for a suitable microkernel technology, the group at CMU had demonstrated a server running on their Mach microkernel which provided a UNIX personality. The OSF BSD server was originally

[4] This section contributed by A. Pérez, S. Rodríguez, L. M. Muñoz, A. García, M. A. Liébana, L. Prieto, Departamento de Arquitectura y Tecnología de Sistemas Informáticos, Facultad de Informática, Universidad Politécnica de Madrid

[5] OSF is a consortium of computer vendors led by IBM, DEC and Hewlett Packard, and was formed in an attempt to wrest control of UNIX from its then owner, AT&T.

based on the CMU UNIX server, but concerns about security, integrity, resource management, compatibility, performance and maintenance prompted extensive changes to the source. Although the OSF rewrote and redesigned large portions of the BSD server so that the OSF server now is quite different from its CMU counterpart, the microkernel remains unchanged and is identical to the CMU Mach 3.0 microkernel. See [Barbou des Places et al 94] for a more detailed description of the software architecture of the server. Figure 7.5 shows a more detailed picture of what happens when a system call is made by a user process; there are four main software components involved.

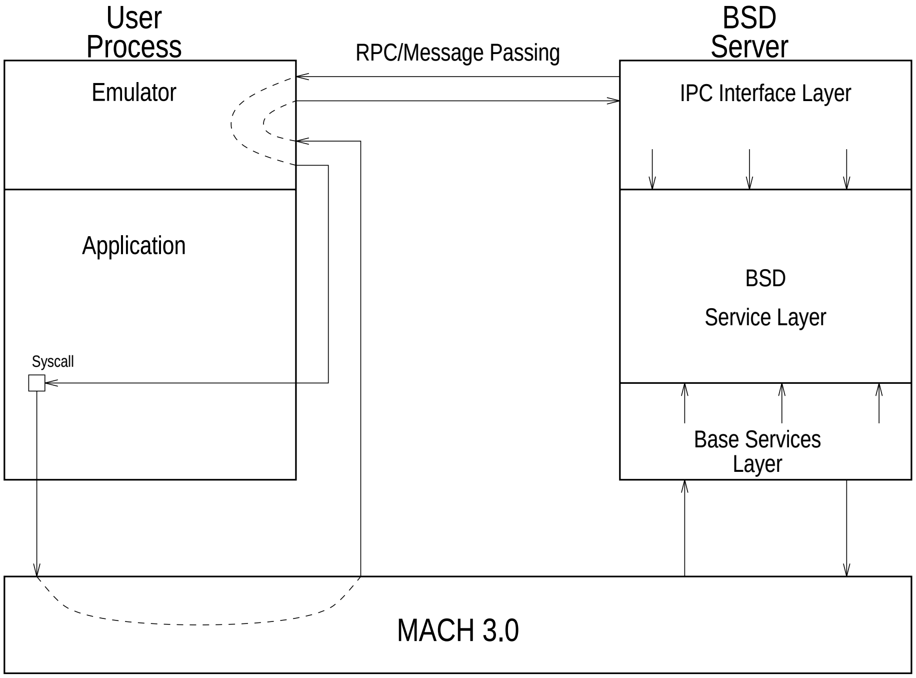


Figure 7.5: Software architecture of BSD server

7.3.1.1 The Emulator

The emulator[6] is a shared library, transparently linked at a fixed point in the address space of each application program, which implements the client side of the OSF1/mk emulation. Access to the emulator is provided by means of the `task_set_emulation()` calls [Loepere 92a]. This dynamic linking of the emulator code to each OSF1/mk process results in binary compatibility with code for other operating systems. When a user process wants a service from the operating system it performs a system call. The library function invoked by the user raises a trap into the microkernel, and the trap is redirected to the emulator via a Mach redirection mechanism. Once in the emulator, the system call is passed to the server using Mach IPC. Although this implies the execution of a system call requires at least two communications, there are some system calls that can be completely executed in the emulator with no communication overhead.

7.3.1.2 IPC Interface Between Emulator and Server Layers

When a system call reaches the server, it expects a message containing the system call number and the parameters of the call. This information is used to get the function in the server that performs the system call. The IPC interface layer is there to hide all the details of the necessary message handling and parameter passing.

[6] Also called the *emulation library*. Note that subsequent releases of OSF1/mk do not rely on an emulation library. When a user program executes a system call trap, the kernel generates an exception message which it sends to a port that the server created to handle system calls. The server then copies in and copies out any data necessary for the call from the calling program's address space. Finally the server returns into the kernel by replying to the exception message, and the kernel returns to the calling program using the information in the exception reply to modify the calling thread's state before resuming it.

There are several types of system calls depending upon the way the communication between the emulator and the server is done. Section 7.4 describes the different types of system calls. The communication mechanisms used are:

RPC Mach provides a RPC interface generator, called MiG, for passing parameters between tasks.

Message Passing Mach message passing is used in most system calls.

Shared Memory When there is a big buffer to transfer between the server and the emulator, the Mach virtual memory mapping mechanism is used to maintain a shared memory zone between the emulator and the server.

When invoked, the IPC interface goes through the following steps:

(1) Identify the calling process.

(2) Activate a thread for each request that has been received.

(3) Call the corresponding system-call-specific routine.

7.3.1.3 BSD Services Layer

This layer implements the semantics specific to the OSF1/mk system. Two parts can be identified in it:

(a) Mapping of the basic system objects onto the Mach objects: as an example, a Unix process is implemented using a Mach task with a thread in it.

(b) Implementation of the operating system services: as far as possible, this part reuses the code of the previous kernel versions as, for example, in the case of the file system or the networking code.

7.3.1.4 Base Services Layer

This layer supplies the low level functions necessary to allow the reuse of large portions of the previous versions of the kernel code. This is necessary because when transforming a monolithic operating system into a user mode server, some low level internal services can no longer be accessed directly. Examples of such services are the synchronization primitives and interrupt masking routines.

Device drivers are another example. They remain part of the Mach 3.0 kernel and therefore cannot be accessed directly by a user mode operating system server. Instead, the base services layer uses Mach IPC to provide the necessary interface between user level operating system code and kernel device drivers.

7.3.2 Source Tree of OSF1/mk

The main directories of the source tree provided in Version 4.1 of OSF1/mk, as shown in Figure 7.6, are as follows:

(a) `export`: Contains the header files and the libraries that are built in the `obj` directory and are copied into the `export` directory.

(b) `obj`: Contains the objects and executable files of the Mach kernel and/or the server after their compilation.

(c) `tools`: Contains the compiler (gcc) and related tools (C-preprocessor, linker, etc).

(d) `src`: Contains all source files of the Mach kernel, the server, the libraries, and the commands.

(e) `src/sbin`: Contains the source files of the system commands not using the shared libraries. These commands are very important in single-user mode or when recovering from an error during shared libraries preload.

(f) `src/setup`: Contains all the shell scripts that build the different components of OSF1/mk.

(g) `src/kernel`: Contains the source files of OSF1/mk.

(h) `src/mach_servers`: Contains the source files of the server and the emulator of OSF1/mk.

(i) `src/mach_servers/emulator`: Contains the source files of the emulator.

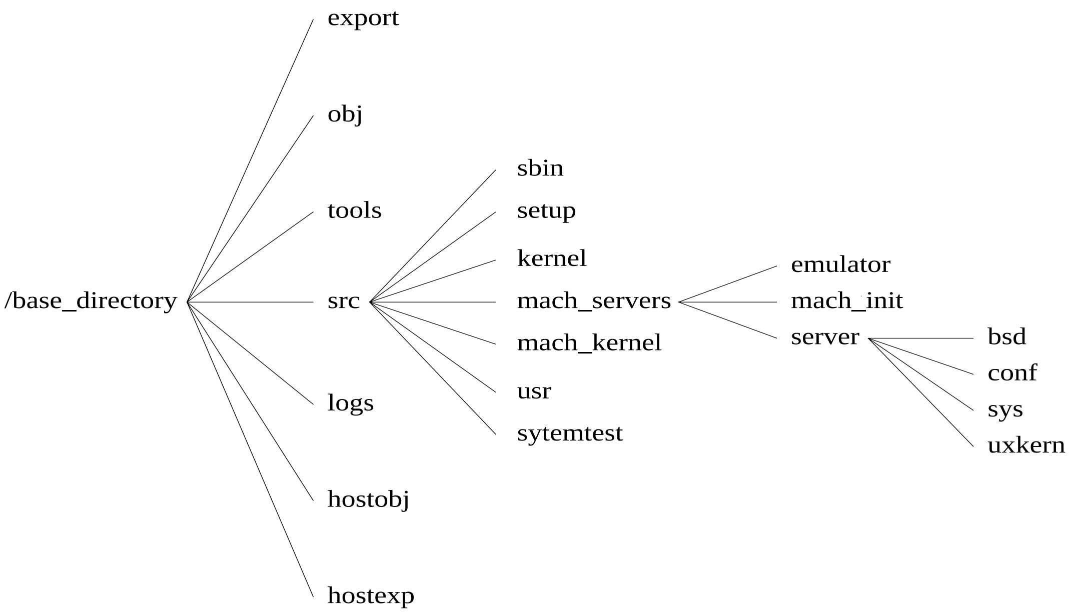


Figure 7.6: Source tree of OSF1/mk Version 4.1

(j) `src/mach_servers/mach_init`: Contains the source files of a server that creates ports for globally interesting services, and hands the receive rights to those ports (i.e. the ability to serve them) to whoever asks.

(k) `src/mach_servers/server`: Contains the source files of the server.

(l) `src/mach_servers/server/bsd`: Contains the source files of the server that implements the POSIX 1003.1 services.

(m) `src/mach_servers/server/conf`: Contains all the configuration files relevant to the OSF1/mk version to be built in the next compilation.

(n) `src/mach_servers/server/sys`: Contains the header files that are not to be exported to the directory `/usr/include`, that is, should be hidden from the user.

(o) `src/mach_servers/server/uxkern`: Contains the source files related to server thread handling and all the source files for the IPC interface between the emulator and server software layers.

(p) `src/mach_kernel`: Contains the Mach 3.0 source files.

(q) `src/usr`: Contains the source files of all commands.

(r) `src/systemtest`: Contains the source files of the tests to be made of the system. Holds internal and external tests, plus any new tests added to check POSIX 1003.4 correspondence.

(s) `logs`: Contains the log files of previous compilations. It is important to maintain previous logs to compare them with logs from new compilations in order to check if the compilation has been successful.

(t) `hostobj`: Contains object and executable files of system commands after their compilation.

(u) `hostexp`: Contains libraries and header files to be exported to system directories. For example all libraries in this directory are the last compilation of libraries in the `/usr/lib` directory.

7.3.3 Monoprocessor & Multiprocessor Configurations

The installation kit, provided by OSF, includes all the binaries and source code needed for building and running OSF1/mk: the Mach and BSD server source files for different platforms (i386, Encore MultiMax and MIPS), the source files of all the applications and system commands, and the source files of the tools used for building the system. Here we will examine the OSF1/mk system for two different installations, one a multiprocessor, the other a uniprocessor. Both installations are for machines based on the Intel i486 microprocessor. They share the same source code, and only differ in the binaries obtained from building OSF1/mk.

7.3.3.1 Corollary 486/smp multiprocessor

The Corollary 486/smp [Corollary 92] belongs to a symmetric multiprocessor family that is based on the Intel i486 processor. It features a tightly-coupled shared memory architecture, in which multiple processors have access to common memory. To prevent memory access from creating a bottleneck, there is a cache on each CPU board. As indicated in Section 6.3, the system uses a dual-bus architecture, featuring a standard EISA bus for peripherals and a propietary 32-bit bus, the Extended C-bus, for processor and memory traffic. This results in a system that preserves compatibility with EISA and ISA peripherals yet achieves a performance level only possible with a bus designed specifically for multiprocessing.

A typical server configuration for compiling, testing and debugging might have 4 processors, 32MBytes of shared main memory, a controller for SCSI hard disk devices and diskette drives, an *Ethernet* interface, and finally, a low resolution graphics controller for the console.

7.3.3.2 i486 PC uniprocessor

A typical i486 PC workstation might have an Intel i486 processor, an EISA bus, 32MBytes of memory, a controller for SCSI hard disk devices and diskette drives, an *Ethernet* interface and a high resolution SVGA graphics controller.

7.3.4 Compilations: Scripts, Makefiles & Configurations

The process of building the OSF1/mk operating requires the building of all the components of it. Those components are:

(a) Mach 3.0 kernel

(b) OSF1/mk server

(c) User Libraries

(d) User commands

The compilation is a complex process handled by the Unix `make` utility. All the environmental variables that `makefiles` expect to be set, are initialized through shell scripts. Those scripts can be found in the directory `src/setup` under the subdirectory corresponding to the platform we are building for: `AT386` (the i386 reference platform). Next the most important shell scripts are described:

(a) `AT386/setup.sh`: Builds the tools necessary to compile the rest of the tree (compiler tools are not included).

(b) `AT386/OSF1MK.sh`: Builds the whole OSF1/mk operating system: All libraries and commands, the Mach 3.0 kernel, the server and the emulator.

(c) `AT386/mach_kernel.sh`: Builds the Mach 3.0 kernel.

(d) `AT386/mach_servers.sh`: Builds the server and the emulator.

(e) `AT386/onecmd.sh`: Builds include files, libraries and commands in the tree, that are specified as parameters (e.g. `onecmd.sh <directory> <target>`).

The shell scripts must be invoked from `src/` directory and with the `sh` as a command interpreter. For example to compile the server and the emulator the command is:

```
sh -x setup/at386/mach_servers.sh >& ../logs/mach_servers.log
```

When executing these scripts it might be a good idea to redirect the standard output and the standard error to a log file (as in the example above) in order to have a trace in case any problem is encountered.

The microkernel, server, emulator and mach_init binaries built by the above scripts may be found below in the following paths:

```
~/obj/at386/mach_kernel/${MACH_KERNEL_CONFIG}/mach_kernel
```

```
~/obj/at386/mach_servers/server/${CONFIG_OPT}/vmunix
~/obj/at386/mach_servers/emulator/emulator
~/obj/at386/mach_servers/mach_init/mach_init
```

`MACH_KERNEL_CONFIG` and `CONFIG_OPT` are environment variables that may be set to select the wished configuration for either the microkernel or the server. They have to be set before using the above mentioned scripts.

The `CONFIG_OPT` environment variable indicates the configuration file used when building the server (`server/conf/AT386/${CONFIG_OPT}`). It is set in the `AT386/host.sh` script (see below), that is read in by the other scripts to set up the basic variables used by all the scripts on the Intel i486 platform.

```
if test $MULTI
then
        CONFIG_OPT="MULTI_FASST"
else
        CONFIG_OPT="DEB_FASST"
fi
```

(a) `DEB_FASST`: the configuration file to compile for the monoprocessor platform PC i486. The number of CPU's `cpus` is set to 1.

(b) `MULTI_FASST`: the configuration file to compile for the multiprocessor platform Corollary 486/smp. The number of CPU's `cpus` is set to 4.

The `MACH_KERNEL_CONFIG` environment variable indicates the configuration file used when building the microkernel (`mach_kernel/conf/AT386/${MACH_KERNEL_CONFIG}`).

(a) `STD+WS`: the configuration file to compile for the monoprocessor platform PC i486.

(b) `STD+COROLLARY+6+XCBUS+BULL`: the configuration file to compile for the multiprocessor platform Corollary 486/smp.

7.3.5 Booting the built OSF1/mk

There are two ways in order to boot the built binaries:

(1) Specifying a boot prompt:

It is necessary first to copy the built `mach_kernel` into `/mach_kernel.new`, the built server vmunix and its emulator into the `/mach_servers` directory as `vmunix.new` and `emulator.new`, for instance, respectively. At the boot prompt, the following should then be entered:

```
mach_kernel.new:/mach_servers/vmunix.new -e /mach_servers/emulator.new
```

(2) As the default boot:

It is necessary to copy the built `mach_kernel` into `/mach`, the built server vmunix into `/mach_servers/startup` and its emulator into `/mach_servers/emulator`.

7.3.6 Debugging a Server

The GNU `gdb` debugger is a symbolic debugger that will be used to debug a OSF1/mk server when it is run as a second server (`/usr/local/gnu-tools/bin/gdb4.6`); however this implies some specific machine set-up that will be described later in this section.

The figure 7.7 shows the execution environment of a second server. The second server runs as a child process of the first server, and can be debugged with a special version of gdb, `gdb4.6`, which is provided with the OSF1/mk sources, and which can debug multiple-thread Mach tasks.

The second server interacts with the first server in order to write the messages in console. The second server console is the first server terminal from which the second one started its execution. There are two different families of processes, which will not be allowed to interact using normal Unix system calls: the first server only knows the server being debugged, not its descendants, and the second server uses a few services from the first one, namely for the console emulation for instance.

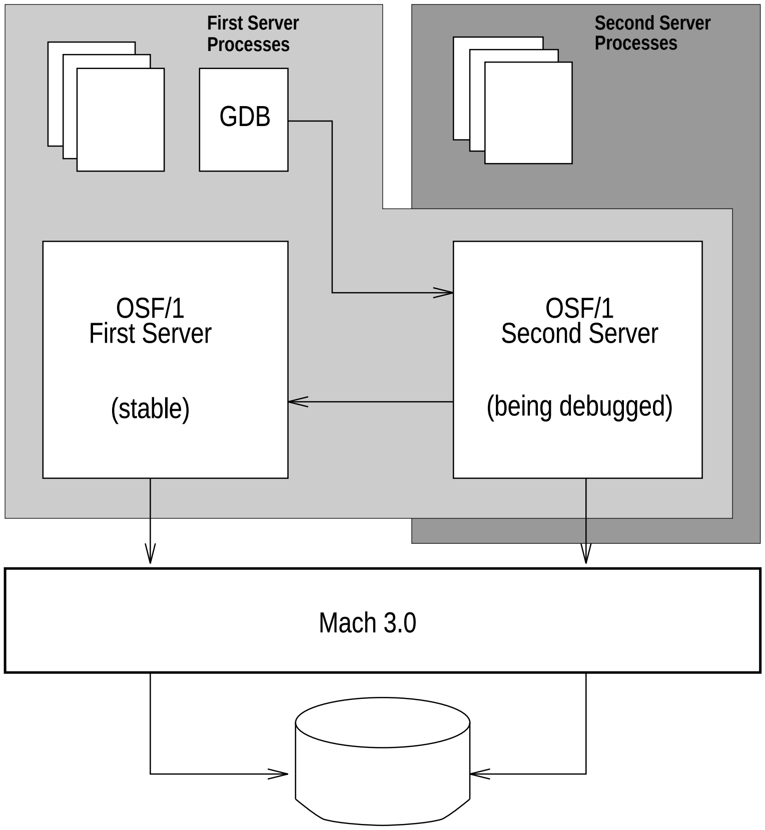


Figure 7.7: Execution environment of the second server

To use the facility of running and debugging a second server it is necessary to establish an appropriate environment. This is, to partition the disk with two different roots (for booting first and second servers), two paging files, `/var` and `/swap`. See the table below, figure 7.8:

FIRST SERVER	**SECOND SERVER**	
sd0a	**sd0g**	***root***
sd0b	**sd0f**	***paging file***
sd0e	**sd0i**	***/var***
sd0h	**sd0j**	***/swap***

Figure 7.8: Partition Table

All read-only file systems, such as `/usr`, can be shared between the two servers because it contains the commands that will not be changed.

7.3.6.1 Running the Second Server

The `~/bin/second` program starts the execution of the second server. The `gethostname()` function gets the name of the local host, but unfortunately the name of the machine in which the second server must run is hardcoded into the program.

For a machine with a different name it will be necessary to change and to recompile this program, and then to establish the ownership and rights to be as follows:

```
-rwsr-x---   1 root     fasst      77470 Jul 08 06:42 second
```

When run, ~/bin/second mounts the root partition of the second server (/dev/sd0g) into the /mnt directory to copy the built emulator into the /mnt/mach_servers directory as emulator.new, and then unmounts the sd0g partition. Finally, it runs the second server, executing:

```
~/obj/at386/mach_servers/server/${CONFIG_OPT}/vmunix
-w -e /mach_servers/emulator.new sd0g
```

The argument -e specifies a file for the emulator and -w for wired threads only.

7.3.6.2 Debugging the Second Server

Once the second server is running you may debug it. The ~/bin/debug script first will recognize the machine in which the second server is running in order to know the path of the built server vmunix and change the current working directory to the vmunix directory.

Then execute ~/bin/srv_gdb[7] vmunix and attach to the process ID of the running second server.

7.4 System Call Extensions to a Server

As stated in section 7.3.1 there are different types of system calls depending upon the way the communication between the server and the emulator is performed. The type of each system call is specified in the file server/conf/syscalls.master with one of the following keywords: MSG, STUB, EMUL and RMUL:

(a) MSG System Calls.
This kind of system call is the most common, it performs the communication between the server and the emulator via Mach message passing.

(b) STUB System Calls.
The key characteristic of these system calls is that they receive special treatment in the emulator. For each STUB system call there is one routine in the emulator that do some especial test before communicate to the server. These test check the validity of the system call parameters to avoid the communication with the server if there is an erroneous value. Once all the tests are passed, the communication with the server is performed by RPC or via a MiG interface.

(c) EMUL System Calls.
This type of system calls an the following one are very similar, both can execute completely in the emulator, without communicating with the server. When the call cannot be completed in the emulator the communication with the server is done either by message passing as in MSG calls or by RPC as in STUB calls. An example of this type of system calls is sigprocmask(), the signal mask of the process is maintained in a memory zone shared with the server and the only action to carry out is to change it, so it is not necessary to involve the server in the execution of this system call.

(d) RMUL System Calls.
The only difference between this kind of system calls and the previous one is an extra parameter added to all the system calls of this type. This extra parameter contains the registers of the machine. Obviously the system calls of this type are machine-dependent. An example of such system calls is sigreturn() called when a process exits from a signal handler and the previous stack frame of the process must be restored.

If a new systems call has to be added to the server, this is not a very difficult task, but it does force one to modify several files. As an example of how to do this, below we outline the steps involved in adding mq_open and qseek system calls to the OSF1/mk BSD Server.

[7] gdb4.6 binaries

7.4.1 Adding the `mq_open` System Call

(1) The file `server/bsd/fasst_msg.c` contains the source code of all system calls related to message passing.

(2) A new entry has to be added in the file `server/conf/syscalls.master` :

```
        269     STD     SERIAL          MSG     4 mq_open
```

The meaning of each column is the following:

- **(a)** `number`: system call number that must be in order.
- **(b)** `type`: one of `STD`, `STDALT`, `OBSOL`, `UNIMPL`, `COMPAT`.
- **(c)** `parallelized`: one of `SERIAL`, `PARALLEL` or `UPARALLEL`.
- **(d)** `emulation`: one of `MSG`, `STUB`, `EMUL` or `RMUL`.
- **(e)** `nargs`: number of arguments.
- **(f)** `name`: name of system call routine.

(3) Depending on the compilation debugging option, extra information has to be inserted in one of the following files:

- **(a)** No debugging. The file `server/conf/files` has to contain the following line:

```
        bsd/fasst_msg.c         standard
```

- **(b)** Debugging. The file `server/conf/files` remains unchanged; however, the template file `server/conf/template.mk` has to be modified by adding the following lines:

```
        OBJS = ${OBJS} fasst_msg.o

        CFILES = ${CFILES} bsd/fasst_msg.c

        COBJS = ${COBJS} fasst_msg.o

        fasst_msg.o: bsd/fasst_msg.c \
                     ${C_RULE_1A_DBG} bsd/${C_RULE_1B}; \
                     ${C_RULE_2}; \
                     ${C_RULE_3}; \
                     ${C_RULE_4}
```

(4) It is necessary to create a new file `mqopen.s` in the `src/usr/ccs/lib/libc/AT386` directory, containing the following code in the same way that in the other system calls:

```
        #include        <syscall.h>
        #include        <machine/asm.h>

        SYSCALL(mq_open)
                ret
```

Moreover, `mqopen.o` must be added to the object list in the file `machdep.mk` in the `src/usr/ccs/lib/libc/AT386` directory.

(5) The prototype of the `mq_open` system call is included in the `src/kernel/mqueue.h` header file.

```
#ifndef _KERNEL
#ifdef _NO_PROTO
extern mqd_t    mq_open();
#else
extern mqd_t    mq_open(const char *, int, mode_t, struct mq_attr *);
#endif /* _NO_PROTO */
#endif /* _KERNEL */
```

7.4.2 Adding the `qseek` System Call to Remove the 2GB Filesystem Limit

The UNIX file system implicitly limits the size of file system to 2GB. This is because the `newfs` command that is used to create a new file system relies on the `lseek` system call to write file system metadata onto the raw device. The `lseek` system call takes a 32 bit offset which determines where future operations will take place. When the file system is being written this offset is a byte offset on the raw disk, and it is this quantity which introduces the limitation on file system size. The file system can support large file systems, such as are required for sensible use of the Stable Disk, but the standard `newfs` does not allow them to be created. To work around this problem a new system call has been added to OSF1/mk: the `qseek` takes a 64 bit quantity as an offset, and `newfs` is then altered to use this call. Once large file systems can be created, the existing file system code can handle them, since it uses offsets referring to disk blocks rather than bytes.

7.5 Driver Extensions to the Microkernel[8]

The development of device drivers is an unpleasant task for a number of reasons. Typically device drivers are implemented as part of the kernel of the operating system. This means that each time the device driver is modified the kernel must be recompiled and the machine rebooted before the new version of the driver can be tested. This need to reboot increases the time taken through each iteration of the modify-compile-test cycle and also reduces the usefulness to other users of the machine on which the driver is being developed.

7.5.1 Developing Device Drivers in User-Space

The above difficulties are primarily due to the driver being part of the kernel. Many of these problems disappear if the kernel provides services which allow user-space programs to provide the functionality of a device driver. The designers of the Mach 3.0 I/O sub-system went part of the way to providing such a facility, and the extensions discussed here attempt to carry this work through to its logical conclusion by allowing drivers to be developed as normal application programs which can be recompiled and run without rebooting the machine. This dramatically reduces the time taken to test modifications to drivers, allows other users to continue using the machine during driver development, and facilitates the use of the more sophisticated debugging tools available in user space.

To support user space device drivers which are independent of kernel code, three new facilities have been added: a service to vector interrupts into user space, a means of mapping a specified portion of physical memory into a task's address space, and a mechanism for dynamically adding a device driver to the system. In some multiprocessor configurations, not all the CPUs have access to the I/O bus. A facility has been added to allow threads which require access to the I/O bus to bind themselves to an I/O-capable CPU, ensuring that they are only scheduled to run on that processor.

7.5.2 The Spy Device `/dev/spy`

To facilitate the memory mapping and interrupt vectoring modifications to OSF1/mk, a new pseudo-device `/dev/spy` has been added to the system. Adding a pseudo-device is a less complex and more flexible method of extending the system than adding system calls. The name `spy` was inspired by the `vspy` system call which serves a similar purpose in GENIX [National 85].

7.5.2.1 Mapping Physical Memory

OSF1/mk allows limited access to the I/O space of the i486's physical address space through the pseudo-device `/dev/iopl`. While this is sufficient to control many I/O devices it falls well short of what is needed to control the Stable Disk. The Stable Disk's registers are mapped into EISA memory which in turn is mapped to a fixed address range in the i486's physical memory map. To access these registers, the spy device has to allow the appropriate portion of physical memory to be mapped into the device driver's task.

To map a portion of physical memory, the task first opens the spy device using the standard UNIX `open` call. Next, the task calls `mmap`, a call normally used to map files into memory. When `mmap` is passed the file descriptor of the spy device, its arguments take on the following meaning:

```
mmap(addr,len,prot,flags,file_descriptor,physical_addr)
```

where `addr` is the location within the task's virtual address space where the physical memory is to be mapped; `len` is the length of the memory range in bytes. The arguments `prot` and `flags` retain their standard meanings,

[8] This section contributed by Jeremy Jones and Danny Keogan, Department of Computer Science, Trinity College Dublin

details of which can be found in mmap(2). The file_descriptor argument is the file descriptor of the spy device and physical_addr is the start address of the range in physical memory to be mapped.

The implementation of this call is made difficult by the layers of encapsulation in the microkernel. It is not possible for the server to create a direct mapping between a region of virtual memory and a region of physical memory, because the microkernel's virtual memory interface does not allow access to a task's page tables from outside the microkernel. This means that part of the implementation of the spy device has to be in the server and part in the microkernel.

Under the Mach virtual memory system, the microkernel allows for a region of memory to be backed by a memory object. A memory object is an abstraction which encapsulates data storage. When a fault occurs in a region of virtual memory, the microkernel looks up the memory object that is backing that portion of the task's address space and sends it a message requesting it to supply the data that ought to be at that location. If, for example, the region of memory is backed by the default pager, it will read the data from disk and supply it to the microkernel. The advantage of this system lies in allowing the kernel access to data from heterogeneous sources, through a single abstraction, that of virtual memory. The data supplied by the memory object may come from disk, the memory of a remote machine or, as in this case, from a particular area in physical memory.

When mmap is called on the spy device, the server creates a new device pager memory object, connects it to the appropriate region of virtual memory in the calling task's address space and returns. As the pages of the mapped region are accessed, they cause page faults which result in messages being sent to the device pager requesting it to provide the data. The pager then invokes the mapping routine of that portion of the spy device that resides in the microkernel. Since this routine is in the microkernel it can make the appropriate changes to the faulting task's page tables and return.

7.5.2.2 Interrupt Vectoring

The standard support for user space device drivers in OSF1/mk requires that each driver have an interrupt handler routine in kernel space. This routine saves volatile registers in a page shared between the kernel and the driver, acknowledges the interrupt and resumes the previously suspended user driver with a call to thread_resume. The driver then processes the interrupt and, when it has no further work to carry out, suspends itself with a call to thread_suspend until another interrupt occurs.

Although interrupt handlers do not require much code, writing one for each user space driver is tedious. A cleaner and more general solution to this problem is to provide a mechanism which allows the user space driver to acknowledge its own interrupts. The spy device offers this functionality through an ioctl call. The syntax for the call is:

```
ioctl(file_descriptor, request, argp)
```

where file_descriptor is the file descriptor of the spy device obtained from a standard UNIX open call; the value passed for request should be SPY_WFI, defined in spyio.h; the argp argument should be a pointer to a struct spy_wfi which is defined, in the same header file, as:

```
struct spy_wfi {
    int hwlevel;
    int swlevel;
    int timeout_ticks;
}
```

The field hwlevel refers to the i486 interrupt level, swlevel is a value used by the kernel to attach a software priority to interrupts, and timeout_ticks is a timeout value measured in milliseconds.

When the ioctl call is executed, the calling thread is blocked until such time as an interrupt occurs or the call times out. The timeout is necessary because the microkernel will not allow a task which has a thread blocked in a system call to exit. The call will fail if the specified hardware interrupt level is already being used.

The call works by allowing a user space task to alter the tables used by the microkernel to keep track of interrupt handlers. Three tables (iunit, intpri, and ivect) keep track of the hardware level, the software level and the address of a handler, respectively, for each available hardware interrupt. The call first checks to see if the given hardware level already has a handler defined, in which case it fails, returning an error value; otherwise, it inserts details of its own generic handler spyint into the tables, while noting the thread identifier of the caller in a shadow table which it maintains. The calling thread is then blocked. When an interrupt occurs, the spy interrupt handler spyint is called and looks up the shadow table, to find the appropriate thread and wake it up.

7.5.3 Dynamically Adding Device Drivers

The standard support for user-space device device drivers in OSF1/mk does not support dynamically adding device drivers to the system. This means that user-space drivers must be started as part of the boot process. To test a modification to a such a driver, a time consuming reboot must be carried out to start the new version of the driver. A new system call `user_device_check_in` was added to the system to allow the dynamic addition of device drivers. This call maintains a table, which associates user space device driver names with the Mach ports to which operations on these devices should be sent.

In most cases, when the server receives an `open` system call, it simply calls `device_open` on the microkernel master device port, which returns a *send* right granting access to the port controlling the device. This sequence has been modified to enable the `user_device_check_in` call to work as follows: where the server would normally call `device_open`, it instead calls a routine `tcd_device_open`, which consults the table maintained by `user_device_check_in` to see if the device is a user space device driver.

If the device is not found in the table, `device_open` is called normally on the kernel's master device port. If the device is found in the table, then `device_open` is called, but not on the kernel master device port; it is called on the master port of the user space device driver retrieved from `user_device_check_in`'s table. Once the device is open, the server caches the *send* right to the device's port obtained from the `device_open` call. This *send* right is indistinguishible to the server from a *send* right that it would obtain a kernel driver, so no further modifications have to be made to any other device operations for it to work with user space drivers.

The server looks up devices in its device switch table based on their major device number to find the appropriate driver name for the device. This is a static structure in the kernel source, and therefore cannot be extended dynamically. Dummy entries are made in this table for each user space device driver. To speed the operation of `tcd_device_open`, a naming convention has been adopted for these internal names for user space device drivers (which are independent of the names which appear in the `/dev` directory): user space device drivers names begin with an asterisk. If the name does not begin with an asterisk, `tcd_device_open` assumes the driver is a normal kernel driver. Altering the kernel device switch table is the only modification of kernel source necessary to implement a user-space device driver using the framework created by these modifications.

7.5.4 Binding Threads to I/O Capable CPUs

The Mach microkernel provides support for the multiprocessor architecture manufactured by Corollary Inc. In the original Corollary machines, which incorporate a proprietary cache-coherent bus, the C-Bus, only one CPU, the base CPU, has access to the EISA I/O bus. With Mach, the first time a thread executes an I/O instruction, it traps into the microkernel. If the thread has sufficient permissions to execute the instruction, the microkernel binds the thread to the base CPU, i.e. it ensures the thread will only be scheduled to run on the base CPU. Here the target machine is a later model that incorporates an Extended C-Bus (or EC-Bus). This provides support for symmetric CPU boards; boards which can access the EISA bus directly. To make maximum use of these boards, a change has been made to the behaviour of the trap handler that is called when a thread executes an I/O instruction. This routine now binds I/O threads in round-robin fashion such that they are spread over all the I/O capable CPUs.

When mapping EISA memory into a task address space using the spy device, no I/O instructions are used. Each thread which needs to access the spied portion of EISA memory must therefore execute a dummy I/O instruction to ensure that it has access to this memory, otherwise it will read random values. This instruction can be any I/O instruction which has no side effects. This arrangement has inspired a further enhancement to the I/O instruction trap handler: to allow the thread to choose which CPU it is to be scheduled on. If the instruction executed is an *inb* instruction and the I/O address is between 0x200 and 0x20e, the trap handler will attempt to schedule the thread on the CPU number indicated in the 4 least significant bits of the address. If this CPU does not exist, the handler falls back on the round-robin system. The round-robin system is always used if the address is 0x20f. Two macros provide a syntactic sugar for this mechanism. `BIND_TO_CPU(x)` will attempt to bind the thread to CPU number x, where x is in the range 0 to 14. `BIND_TO_SYMMETRIC_CPU` will bind a thread to a symmetric CPU in the round-robin fashion mentioned above.

7.6 Fault Tolerance Extensions to the Microkernel[9]

By now we have a good idea of the fundamentals of Mach, and at last can speculate on what enhancements are needed for it to conform to the principles of Section 7.1. The aim is to provide a fault tolerant virtual machine (see Figure 7.9) through the use of hardware and the seed, or an equivalent fault tolerant microkernel. This virtual machine is to have no single point of failure and it should gracefully degrade for multiple points of failure. If any one hardware component fails, the virtual machine is to recover transparently (although it need not guarantee recovery from multiple hardware failures).

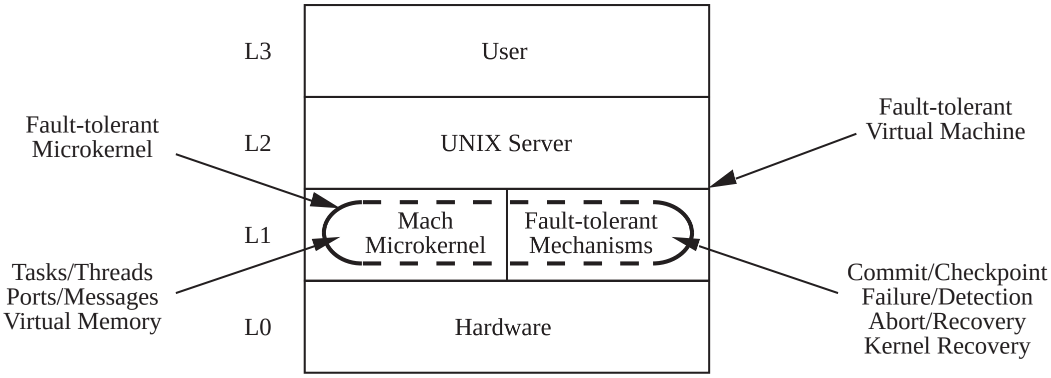


Figure 7.9: Virtual machine

We know that in order to ensure that applications continue to run in the face of failure, enough information about the state of the system must be saved to ensure that if one hardware failure did occur, the most recently saved system state could be restored and execution of the application could continue from there. We also know that the system state is comprised of all the individual processor states, and that when recovery is carried out at the processor level the system is not concerned with the individual state of the currently executing applications, it is concerned with the system and hence processor states at particular instances of time (the *recovery points*).

7.6.1 Fault Tolerant Mechanisms required in the Microkernel

The fault tolerant microkernel has to use the mechanisms provided by the underlying hardware to ensure that the above holds true, i.e. that if a processor fails, its processes should be recovered in a fashion transparent to the user. It does this by using *processor transactions*.

7.6.1.1 Processor Transactions

A processor transaction is used to provide backward error recovery in the event of a hardware failure. Each processor is part of one and only one processor transaction. The microkernel must be able to *create*, *commit* and *abort* a processor transaction. Here the creation and committment of a processor transaction are combined into one act called a *checkpoint*.

Processor transactions are initiated on each processor by the committment of the previous transaction and the establishment of the next. They are begun by the microkernel on each processor. As each processor accesses shared memory the processor dependency groups merge and expand, and thus there may be one or more processors as members of such a transaction. The number of members may increase during the life span of the transaction, but if one processor withdraws from a transaction before it completes, then all the remaining members of that transaction must be rolled back.

Each processor must be able to explicitly invoke a checkpoint by issuing the relevant software commands which use the mechanisms provided by the hardware.

While processors are independent of each other, each processor may decide when to checkpoint, without concern for any other processor's checkpoints. If every processor in the system was independent and the system was

[9] This section contributed by Paula McGrath, Department of Computer Science, Trinity College Dublin

rolled back to a consistent state, each processor would have its most recently checkpointed state restored even though these checkpoints may have occurred at different points in time.

Once processors become dependent on each other, by accessing modified shared data, it is not acceptable for dependent processors to independently checkpoint. A restoration of the system state in this case (with each processor restoring its most recently checkpointed state) would not lead to a consistent system state. Processor dependencies must be logged and tracked to ensure that dependent processors are checkpointed together. In this way a consistent state of the system may be restored if necessary.

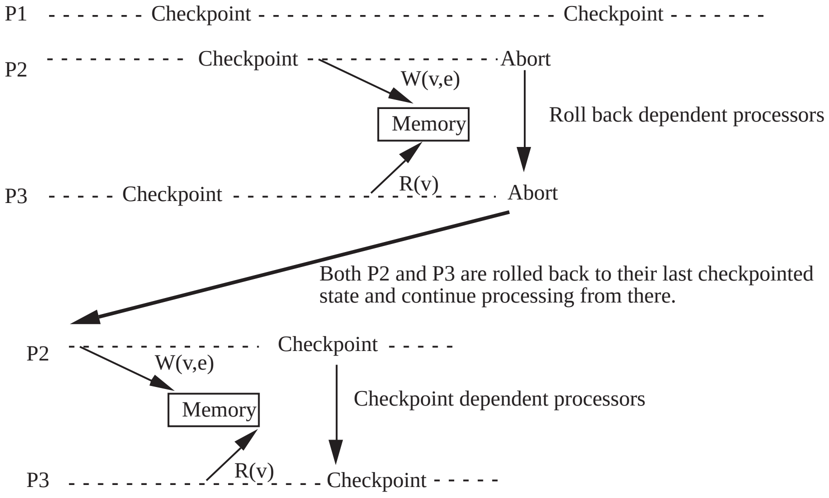

P1
Checkpoint
Checkpoint
P2
Checkpoint
Abort
W(v,e)
Memory
Roll back dependent processors
R(v)
P3
Checkpoint
Abort
Both P2 and P3 are rolled back to their last checkpointed state and continue processing from there.
P2
Checkpoint
W(v,e)
Memory
Checkpoint dependent processors
R(v)
P3
Checkpoint

Figure 7.10: Three dependent processes subject to rollback

7.6.1.2 Processor Dependencies

Processor dependencies are tracked by the SM. Processor dependency groups are created using this information. A dependency is created between two processors when one accesses the same area of shared memory that the other has written to. The dependency tracking is affected by type and order of access. If two processors read the same area of memory no dependency between the processors is created.

Let us take a concrete example. In Figure 7.10 the three processors, P1, P2 and P3, are initially independent. If a rollback occurred, the processors would be rolled back to their most recent checkpointed state. But after P3 reads the value written by P2, P3 becomes dependent on P2. If P2 aborts before its next checkpoint, then P1 must also abort. After the two processors are rolled back, the dependency is created again. This time P2 checkpoints and P3 is implicitly checkpointed. P1 is not affected as it is still independent of the other two processors.

Again let us take a concrete example. In Figure 7.11, P1 and P2 are processors and P1 is the current writer of memory location *v*. If P2 subsequently reads that same memory location, P2 becomes dependent on P1. Similarly if P4 writes to location *u*, P3 becomes dependent on P4. The arrows indicate the dependency relationships.

Actions over time are: P1(w,v), P2(r,v), P3(w,u), P4(w, u) → time

Dependencies relationships are: P2(r) → P1(w) P3(w) → P4(w)

Figure 7.11: Dependency relationships between three dependent processes

A consistent state of the system must be checkpointed where there are processor dependencies. When a processor is to be checkpointed, an implicit checkpoint is issued to all the other processors which are in the same dependency group, and the processors in the descendant dependency groups must also be checkpointed.

In the event of a hardware failure the last saved consistent state of the system is restored and processing continues with the processes unaware of the failure. When one member of a dependency group issues a rollback request, all the members of the group must be rolled back.

7.6.2 Checkpointing

When a checkpoint occurs, the state of a processor is saved. The SM provides the mechanism for checkpointing the state of a processor. It knows what parts of memory have been modified by each processor. However, the state of a processor is everything that has been modified by the processor since the previous checkpoint, including CPU state, registers, cache, etc.. All of this must be saved. As a byproduct, a checkpoint clears a processor of any dependencies.

7.6.2.1 The Checkpointing Algorithm

The checkpointing algorithm is implemented in software using the mechanisms provided by the hardware (see Chapter 4.1). There are two phases to the hardware part of the checkpoint and one phase for the microkernel. One of the processors is the *Initiator*. It is responsible for coordinating the processor checkpoint; it does not have to be the processor which requests the checkpoint. The other processors fall into one of two categories: (a) member of the dependency group, (b) independent of the dependency group. These two categories are mutually exclusive. There may be zero or more entries in either category. In the following paragraphs a description is given of the tasks of the Initiator, the other processors and the SM (also see Figure 7.12).

The Initiator

(a) Request the other processors to *Stop all Activity*

(b) Read the dependency matrix from the Stable Memory Unit(s)

(c) Generate the Dependency Group

(d) Request the processors which are not in the Dependency Group to *Continue Processing*

(e) Tell the processors which are in the Dependency Group to *Prepare to Commit*

(f) Wait for the Dependency Group to signal their acceptance of the message

(g) Inform the Stable Memory Unit(s) to *Prepare to Commit*

(h) Request the processors which are in the Dependency Group to *Continue Processing*

(i) Request the Stable Memory Unit(s) to *Commit*

The Other Processors

(a) On receipt of a *Stop All Activity* command save the registers

(b) Wait for the next command

(c) If command == *Continue Processing* Then
    Resume processing

(d) If command == *Prepare to Commit* Then
    Save Registers and clear/flush cache
    Wait for *Continue Processing* command

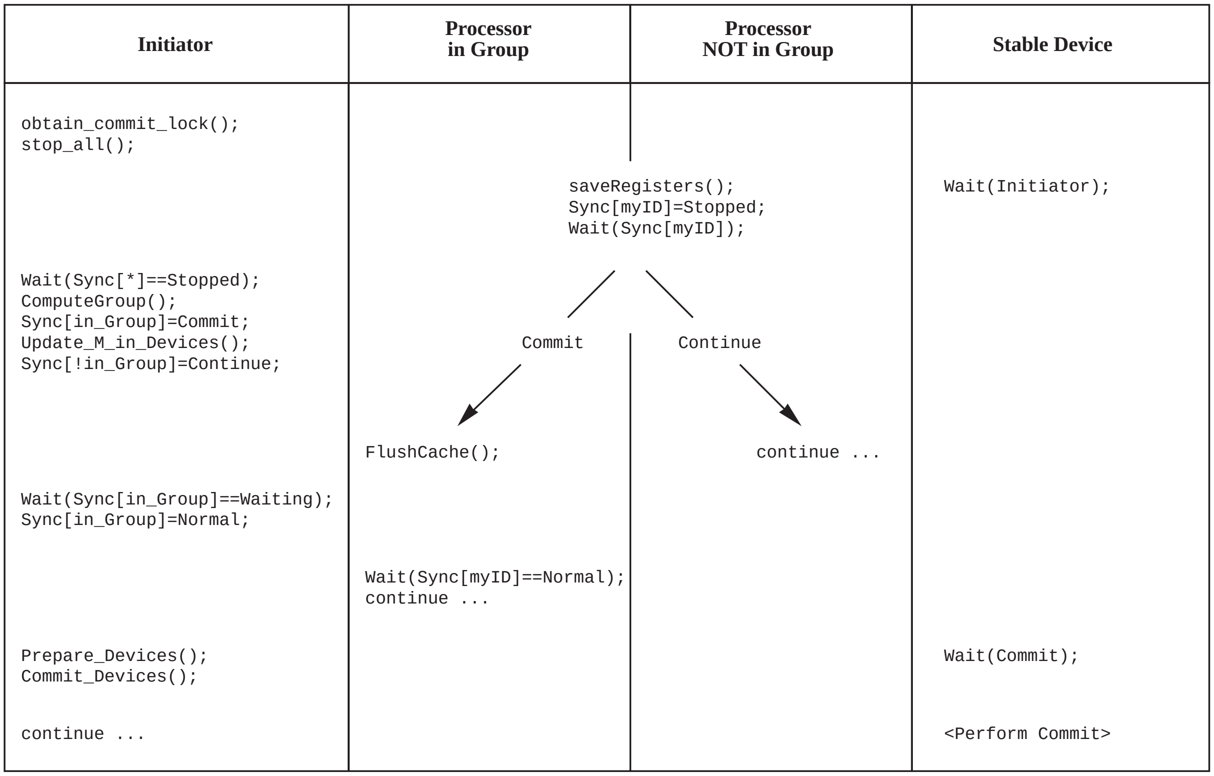
Stable Memory Unit(s)

Phase 1 While the checkpoint is active check that the Initiator is still alive
    Pass the Dependency Matrix to the Initiator

Phase 2 *Commit* the Data

The SMs may be considered as two banks of memory, as in Figure 7.13, one containing the current data and the other containing the recovery data. When the memory *commits* the data is copied from the current memory bank, *bank1*, to the recovery memory bank, *bank2*. When the memory *aborts*, the data in the recovery *bank2* is copied to the current *bank1*.

To summarise, when a checkpoint is taken the state of a processor is saved. The checkpoint is taken by the microkernel using the mechanisms provided by the SM.



Initiator	Processor in Group	Processor NOT in Group	Stable Device

obtain_commit_lock();
stop_all();			

	saveRegisters();
	Sync[myID]=Stopped;
	Wait(Sync[myID]);
			Wait(Initiator);

Wait(Sync[*]==Stopped);
ComputeGroup();
Sync[in_Group]=Commit;
Update_M_in_Devices();
Sync[!in_Group]=Continue;
	Commit	Continue	

	FlushCache();	continue ...	

Wait(Sync[in_Group]==Waiting);
Sync[in_Group]=Normal;			

	Wait(Sync[myID]==Normal); continue ...		

Prepare_Devices();
Commit_Devices();
			Wait(Commit);

continue ...			<Perform Commit>

Figure 7.12: Execution of a checkpointing algorithm

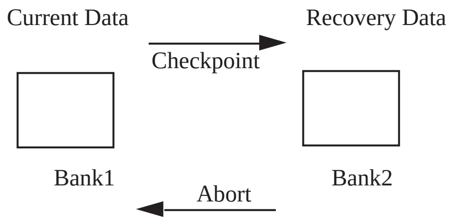


Figure 7.13: Checkpointing and rollback

7.6.3 Rollback

When a rollback is initiated as the result of one hardware failure in the system, it is necessary to transparently roll back a processor and resume execution of the process which was executing at the time of the most recent checkpoint, as though nothing has happened. There may be a problem with rescheduling the process elsewhere in the system because the checkpoint could have occurred at a time which was not convenient for the process; it may not always be feasible to reschedule a process.

Rolling back a processor results in the restoration of the state of the processor and hence a consistent state of the system is also restored. The number of processors rolled back depends on the type of failure. Either all the processors are rolled back or the members of the affected processor dependency groups are rolled back.

Let us again take some concrete examples. In Figure 7.14 we assume for simplicity that checkpoints occur on a context switch. Checkpoints terminate one processor transaction and start another. If there is no rollback both processors continue as normal.

Initially P1 has no dependent processors. If P1 aborts during execution of S2, P1 is rolled back to **Y**; all modifications made in the SM by the processor P1 on behalf of both the microkernel and the process since the

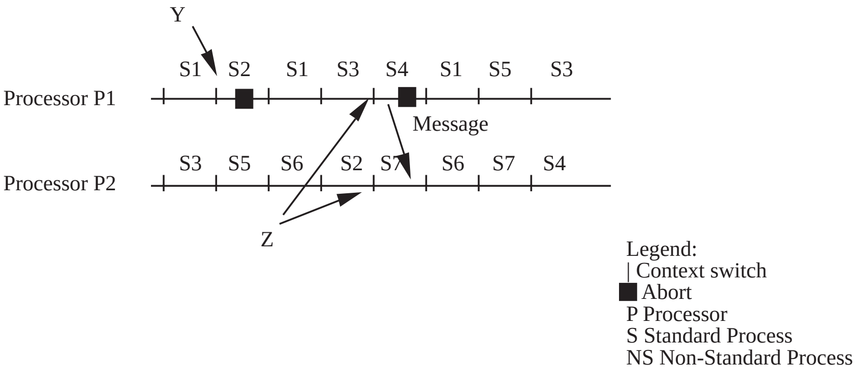


Figure 7.14: Two dependent processes that checkpoint at context switches

checkpoint **Y** are undone by copying the data in the recovery *bank2* to the current *bank1*. Also in this example, when process S7 reads a message from process S4, processor P2 becomes dependent on processor P1. If P1 aborts after S7 reads the message but before its next checkpoint, both P1 and P2 are rolled back to checkpoint Z, causing all the *bank1* data modified by both processors to be overwritten by the recovery data from *bank2*.

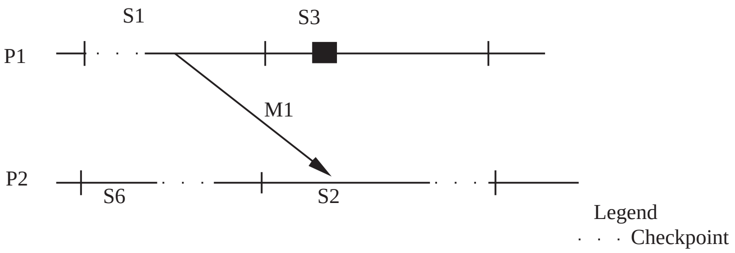


Figure 7.15: Two dependent processes that checkpoint at any time

In Figure 7.15 there are two processors, P1 and P2. In this example checkpointing is assumed to occur at any point in time during a process' execution, i.e. there is no guarantee that a checkpoint occurs at a context switch. P1 checkpoints during the execution of S1 and P2 checkpoints during the execution of S6, where neither occur at a context switch. P2 becomes dependent on P1 by reading the message M1 from S1. P1 aborts during the processing of S3 and both processors are rolled back to their last saved state, which happens to be part way through the execution cycle of the two processes. There is no knowing the state of the processes or what they were doing when their most recent checkpoints were taken.

7.6.4 Standard and Non-standard processes

With the seed we introduced two types of processes, standard and non-standard, and this concept must be extended to the microkernel. A process which accesses both a local state and implicitly recoverable objects is a standard process. A non-standard process is one which accesses objects which are not implicitly recoverable. Objects which are implicitly recoverable are those which are assumed to be represented in the STM. Examples of non-implicitly recoverable objects are I/O devices, and so I/O traffic should be separated and contained in processes devoted to I/O alone.

7.6.4.1 Standard processes

A standard process is unaware of the occurrence of either an explicit or implicit checkpoint or abort. If a processor that is executing a standard process is checkpointed, it should be possible to restart that process from its last checkpoint. This may not be at a point natural to the process but it should be possible. No message is to be lost or handled twice by a standard process if a roll-back occurs. If a sender checkpoints and the receiver rolls back before the message arrives, the microkernel must re-send the message.

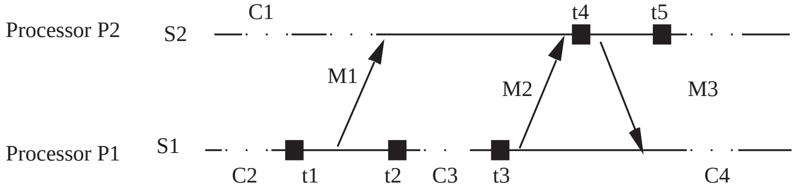


Figure 7.16: Two dependent standard processes

In Figure 7.16 both S1 and S2 are standard processes running on two different processors. No other processes are scheduled on the processors in the given time frame. Initially there is no dependency between them and they both checkpoint within their own time frame. If either of them fails before the message M1 is read and after checkpoints C1 and C2, it will resume execution with the state saved at C1 or C2. After Processor P2 reads the message from process S1 to process S2, P2 becomes part of the dependency group for P1. The dependency is registered the moment the message is read (P1 does not become a member of the dependency group for P2).

If S1's processor fails at time t2, after P2 has read M1 and before checkpoint C3, both processors are rolled back. If P1 fails after checkpoint C3 and before M2 is sent at time t3, only P1 is rolled back because the dependency between the two processors has been cleared. If P2 fails before M3 is read at time t4, only P2 is rolled back.

Time	Dependency Groups	P1 Fails Processors Affected	P2 Fails Processors Affected
t1	None	P1	P2
t2	(P1, P2) $P1 \xmapsto{wr} P2$	P1, P2	P2
t3	None	P1	P2
t4	(P1, P2) $P1 \xmapsto{wr} P2$	P1, P2	P2
t5	(P2, P1) $P2 \xmapsto{wr} P1$	P1	P1, P2

Figure 7.17: Dependency relationships between the processes in Figure 7.16

New dependencies are created by the processors reading messages sent from each other.

7.6.4.2 Non-standard processes

A non-standard process is informed when either an implicit or explicit checkpoint/abort occurs. It does not differentiate between explicit or implicit. Non standard processes must be made aware of these events because they deal with devices which may not be rolled back and have to be handled in a special way.

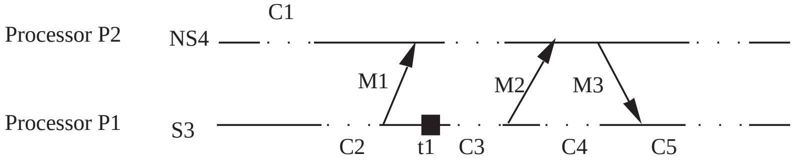


Figure 7.18: Two dependent processes: S3 is a standard process and NS4 is a non-standard process

In Figure 7.18 there are two processes S3 and NS4 running on processors P1 and P2. S3 is a standard process and NS4 is a non-standard process. NS4 is not implicitly recoverable because operations are initiated which can not be rolled back. This causes no difficulties when NS4 is not dependent on S3. After M1 is read, NS4 is dependent on S3. If S3's processor fails before checkpoint C3 occurs and after M1 is sent, S3 is rolled back and an attempt is made to roll back NS4 as well. It is not possible to render void all NS4's operations since its last checkpoint, as it interacts with the external world, and these interactions may not be rolled back.

Nevertheless, the state of NS4 (checkpoint C1) is restored, and an attempt is made to undo any of the changes made to objects in the system. This is possible with STM objects, but the process must also do some extra processing before it is ready to resume operation. This is done by a special routine that is only executed after a processor has been rolled back. There is a corresponding routine that is executed when a non-standard process is checkpointed.

One way of reducing the probability of having to roll back a non-standard process is to checkpoint the processor which creates the dependency immediately after the dependency is created, since after a checkpoint all dependencies created by the checkpointing processor are removed. This is illustrated in Figure 7.19. After reading message M1, processor P2 is dependent on processor P1. Since NS6 is a non-standard process, the dependency should be broken as soon as it has been created, by checkpointing as soon as the message has been read. NS6 will rely on the microkernel to re-send it any messages it loses due to the rollback of processor P2. The failure of processor P1 after checkpoint C3 will have no effect on processor P2 as the dependency has already been cleared. Any dependencies with a non-standard process should be broken as soon as possible. The microkernel has to recognise this and explicitly checkpoint the processor. This is an example of where an explicit checkpoint is used.

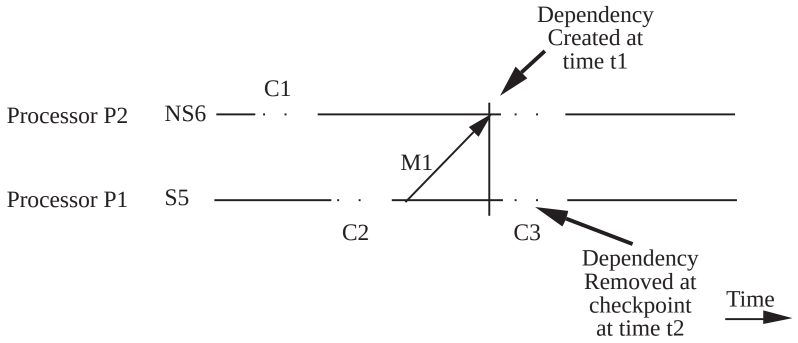


Figure 7.19: A standard and a non-standard process, where dependent processes are checkpointed immediately *after* a dependency is created

Another way of reducing the probability of having to roll back a non-standard process is by checkpointing the processor that could create a dependency before the dependency arises, for example, by checkpointing a processor after every message is sent to a non-standard process. In this way no dependencies will arise from the message being read. In Figure 7.20 Processor P1 checkpoints as soon as a message is sent to the non-standard process. No dependency is created when Processor P2 reads the message M1.

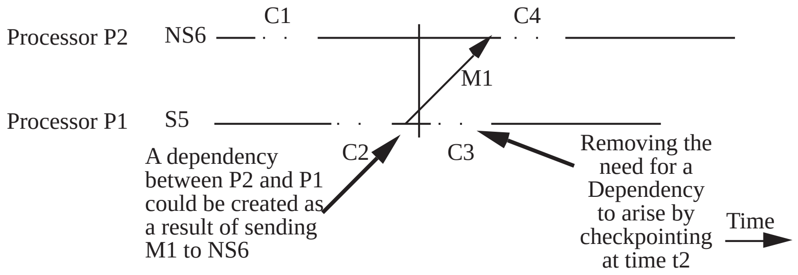


Figure 7.20: A standard and a non-standard process, where processes are checkpointed immediately *before* a dependency is created

7.7 Unresolved Microkernel Issues[10]

Although it all looks very impressive, Mach actually falls well short of what is required for the recovery protocol. Most microkernels suffer similarly. A number of problems remain:

(a) Checkpointing algorithm Difficulties with checkpointing arise where checkpoints happen at unknown points of time. Unresolved issues include what is defined as the start of time, what happens at the start of time, when is the first checkpoint taken ? How is a checkpoint message treated by the processor - is it similar to an interrupt ? Can a processor be checkpointed at all times or are there some problem areas, such as locks ? What happens if there is a crash of the processor which issued the explicit checkpoint - does the Initiator continue with the checkpoint ? What if the checkpoint is only partially done when one of the group's processors dies ?

(b) Rollback For rollback, the major unresolved issue is what exactly is the rollback algorithm ? Also, what happens if there is a failure during checkpointing which causes a rollback ? What if there is a failure during rollback ?

(c) Standard processes Referring to Figure 7.16, if S2's processor fails after M3 is read by processor P1, both S1 and S2 are rolled back to checkpoint C3. As both are standard processes, enough information must be saved when the checkpoint occurs to allow the rescheduling of S2 elsewhere in the system. How to do this is still unresolved.

(d) Non-standard processes For a standard process, rollback is handled by the combination of the microkernel and hardware. The process itself must do some extra processing before it is ready to resume operation. It has a special routine which is only executed after a processor has been rolled back. There is a corresponding routine which is executed when a non-standard process is checkpointed. What exactly these routines do has yet to be specified. A question which also arises here is should non-standard processes be part of processor transactions ? If they are not, non-standard processes should not use stable memory and all their data structures should be stored in non-stable memory. How is a processor initialised to use stable and/or non-stable memory ? Should the memory locations of a non-standard process be recoverable ? A non-standard process can be dependent on standard processes, and vice-versa, but these dependencies should be broken as soon as possible. What about dependencies between non-standard processes ? If the microkernel is aware of the difference between standard and non-standard processes, then non-standard processes should register themselves. Also the microkernel needs a mechanism to store messages which have to be re-sent to non-standard processes. Recovery measures implemented by non-standard processes have no effect on standard processes. There should be a firewall between standard and non-standard processes. Rolling back a non-standard process should not affect any other processes. Another important issue is that within the dependency tracking hardware no dependency is created between processes on the same processor, which is the more usual case for I/O. In this instance perhaps the processor should checkpoint itself at regular instances to avoid having to rollback the non-standard process.

(e) User recovery level User processes need the ability to control their own transactions (let us call these *user transactions*). The fault tolerant microkernel must provide the upper layers with the functions which allow user processes to Begin, Commit or Abort their own transactions. Thus there are two levels of checkpoints needed - those which occur at the processor level and are partially under the control of the microkernel (these may be explicit or implicit checkpoints/aborts), and those which occur at the user level and which are under user control (these will all be explicit). A user level recovery mechanism is required because a user level process cannot rely on processor transactions. Processor transactions are not under user level control; they are used and maintained by the microkernel. Should a processor abort there is no guarantee that the processor (and hence the active process) will be rolled back to its most recent explicit checkpoint. It will be rolled back to the most recent checkpoint, which may be either explicit or implicit, since the life of a user transaction may be longer than one processor transaction. In contrast, when users explicitly commit a transaction, they might expect that any processor transactions running beneath the user transaction will be checkpointed, and similarly they might not expect a processor transaction to last longer than a user transaction. The easiest way to avoid

[10] This section contributed by Brian Coghlan, Department of Computer Science, Trinity College Dublin

confusion is to allow explicit control of the behaviour by the user. For these user transactions, many questions remain unanswered. For example, is another process allowed to read the data involved during the transaction ? How are deadlock and livelock avoided ? Does the Begin command include the data which will be required by the process ? Is two phase commit used for the user transactions ?

(f) Stable Memory failure Failure of the SM is equivalent to the failure of the whole system, since it is used to hold the state of the system, and so it should internally handle as many failures as possible. The extent of this, and whether there is interaction with the microkernel, are as yet undefined.

(h) Processor module failure A processor module failure is equivalent to that module issuing an abort. The failure of a processor is recognised by the other processors and it is handled by the stable memory and an elected processor. The processors are fail-stop, and so if a processor module fails, it is taken out of service immediately. A processor failure must be recognised. One way to detect a processor's failure is by using timeouts or interrupts. There is no actual restoration of interrupted processes as a valid processor state has been saved by the SM. A scheduler runs on each processor and has a local and global queue. The scheduler is unaware of any abort having taken place. The process running at the time of the fault and any others which have been run since the last checkpoint are rolled back by the restoration of that processor's state and the state of the dependent processors. The microkernel selects one processor to complete the restoration. The elected processor then has to do a number of functions. It must redistribute the failed processor's local scheduler queue if possible [Black 90]. This queue contains processes that are permanently bound to the failed processors CPU. It must also reschedule the process which was running when the last checkpoint was taken. If the checkpoint was taken at a context switch there is no further computing to reschedule a process elsewhere in the system. If the checkpoint was taken while a process was running some further steps may need to be taken. This is still an unresolved issue.

(i) Bus failure Failure of the bus should be recoverable, but may be treated as a catastrophic error.

(j) Stable Disk subsystem failure The Stable Disk interface might offer a number of modes of usage to its users. One might be a regular disk interface and another a disk interface which uses the full potential of the subsystem. When it operates as a regular disk interface it must be a non-standard process. The disk driver provides the interface to the subsystem hardware. Disk errors are expected to be handled by the disk driver, and not to be propagated to any users of the subsystem, such as the file system. It is intended that the file system be a standard process. If the processor that the file system is running on is rolled back, the file system should be unaware of having issued any read or write commands to the stable disk driver since its last checkpointed state, i.e. should be able to assume that any commands that were sent to the stable disk driver up to the last checkpoint will have been carried out, but that any commands issued since then will not have been acted upon. It is up to the stable disk driver to ensure that this is the case. How to do this is still unresolved. Other unresolved issues include whether any of the processor state is stored on physical disks, if it possible to rollback the physical disk, and how paging is effected.

(k) Failure of other I/O devices The failure of any I/O device other than the Stable Disk is likely to be unrecoverable but detectable. Thereafter, the simplest response is to defer use of the failed device until it is restored, but this requires a means of indicating to the rest of the system that this device is out of service, otherwise the system may hang due to the unavailability of the device. Any other response will require a device-specific solution.

(l) Power failure In the event of a power failure, the system must degrade in such a fashion that after power-up user applications may continue as if there was no such failure. All the processors must restart with the state of the processes as they were when the processors were last checkpointed. The non-standard processes will have to perform special actions. Much of this might be integrated into the boot process, but just how is as yet undefined.

(m) Adding new components What happens if new component are introduced to the system ? As yet this is undefined.

This is a formidable list of hurdles, and there is no guarantee as yet that they may be overcome. A number of proposals have been made that might handle the problems. Most have not been tested. As an example, let us look at two such proposals, the first for the boot process, and the second for non-standard I/O.

7.8 Proposal 1 : Boot process handling[11]

The first question posed in Section 7.7(a) above is ”what is defined as the start of time, what happens at the start of time, when is the first checkpoint taken ?” The start of time can be defined by the boot process, which can also invoke the first checkpoint. In fact, these are not the only issues that may need attention during booting (for example, see 7.7(l) above). Before attempting to tackle these problems, let us outline just what happens during a Mach boot.

7.8.1 Mach booting

The Mach bootstrap starts at a special machine-dependent entry point (*i386* for the i486 processor). Firstly, all necessary CPU and memory initializations are done, including a call to `pmap_bootstrap()`, then the first hardware-independent Mach function `setup_main()` is given control. The first processor that gets the *start_lock* becomes the *master-CPU*, while the remainder call `slave_main()`. The flow of control is shown in 7.21.

This first Mach function then calls the initialization routines of the basic Mach modules:

(a) `panic_init()` resets the internal panic flags to enable one panic message to be issued,

(b) `sched_init()` initializes the basic scheduler variables (for example, the minimum context switch time) and the action and wait queues,

(c) `vm_mem_init()` sets up the virtual memory system by defining the resident memory structures (from this point on only virtual addresses are used),

(d) `init_timers()` initializes the kernel timers and starts the one for the *master-CPU*,

(e) `initc_timeout()` sets up the Mach timeout timers, and

(f) `mapable_time_init()` starts the internal softclock device.

Then, after the virtual memory system is running, the remaining machine-dependent initialization (FPU, devices, etc.) is done by calling `machine_init()`. Back in `setup_main()`, the main Mach subsystems, (IPC, task, thread), are initialized via `ipc_bootstrap()`, `task_init()`, `thread_init()`, `swapper_init()`) and `ipc_init()`, and the timeout-driven routines `recompute_priorities()` and `compute_mach_factor()` are started by calling them for the first time. `recompute_priorities()` updates the priorities of all threads, and `compute_mach_factor()` calculates some load statistics.

At this point, the time has come to carefully create the first thread (*startup_thread()*) and activate it (in `cpu_launch_first_thread()`) by setting the *active* variables and performing a `load_context()` directly. The *startup_thread* executes the routine `start_kernel_threads()` (this has been defined during creation of the *startup_thread*). This function creates the *idle_thread* for all the processors and also creates the following threads within the kernel task:

(a) *reaper_thread* runs, like all the others, forever, and destroys threads on request,

(b) *swapin_thread* supports swapping of threads,

(c) *sched_thread* handles periodic calculations in the scheduler that are not done at interrupt level, and

(d) *action_thread* shuts down processes or changes their assignment.

It then unlocks the *start_lock*, which in fact starts the other processors (`start_other_cpus()`). After starting the user bootstrap `bootstrap_create()`, the *startup_thread* becomes the pageout daemon by calling `vm_pageout()`. The `bootstrap_create()` function creates a *bootstrap_task* and a *bootstrap_thread* to run the `bootstrap()` function, which in turn creates a *user_task* and a *user_thread* to run a startup file to create servers, etc.. The *bootstrap_thread* then becomes the `default_pager()`.

In the case of slave processors, `slave_main()` just calls the hardware-dependent function `slave_machine_init()`, which does the remaining machine-dependent initialization, and then gives control to `cpu_launch_first_thread()`, where a thread is chosen by `choose_thread()` and then started directly via `load_context()` for this slave processor.

[11] This section contributed by Brian Coghlan, Department of Computer Science, Trinity College Dublin, essentially as a precis of [Jöhnk et al 92].

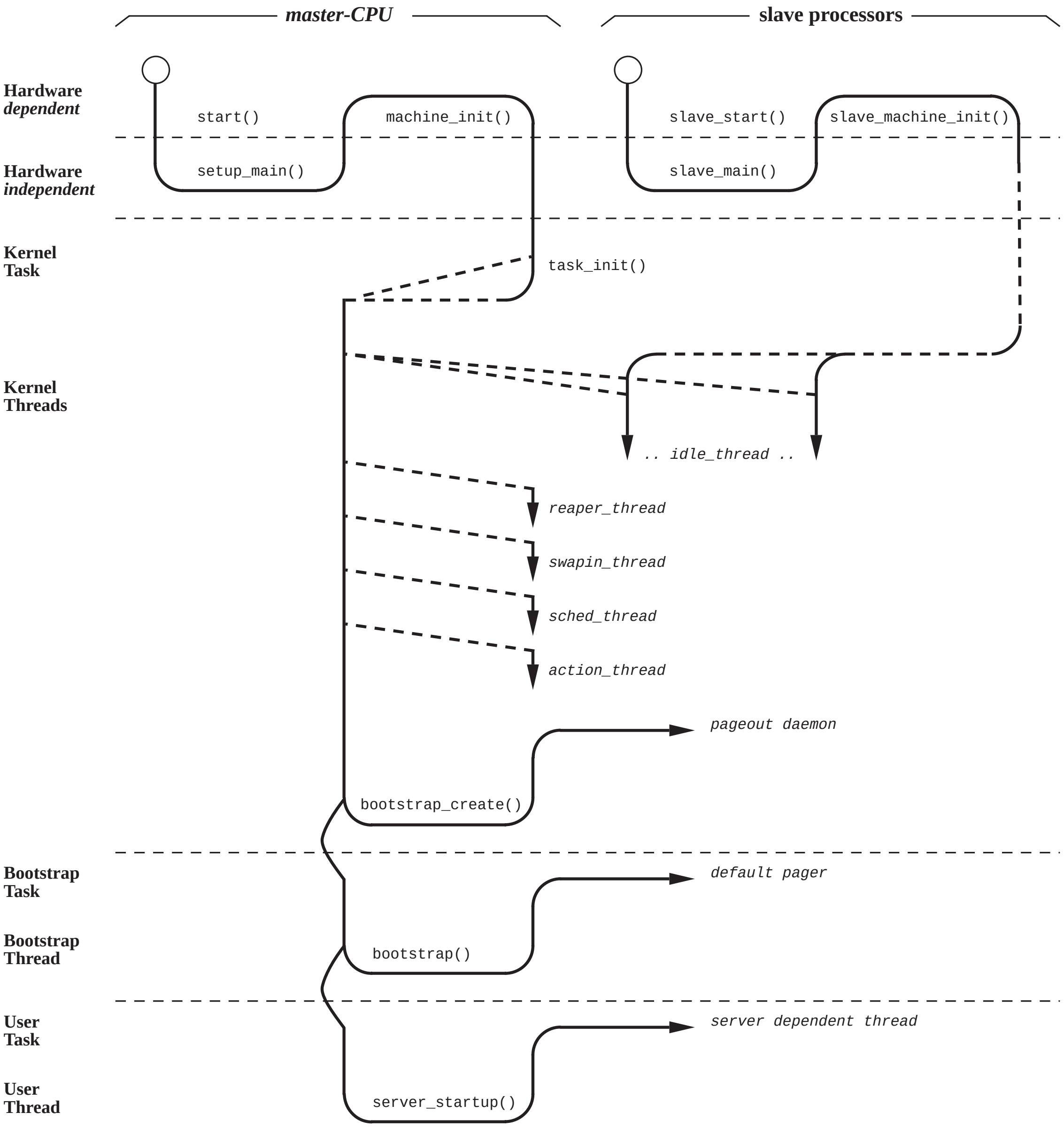


Figure 7.21: Flow of control during a Mach boot

7.8.2 An implementation proposal for boot handling

Now that we understand the boot process, let us consider the extensions that are needed to support the recovery protocol. This has to handle three different situations [Jöhnk et al 92]. Firstly, there is the normal startup outlined above, where an image of the microkernel is loaded from disk and executed. This kind of bootstrap, called a *cold boot*, is done on the very first startup or after regular shutdowns. The situation is different when the system was stopped due to a power failure. In this second case the execution should continue at the last checkpoint taken before the power failure occurred. For this, a *warm boot*, a few suitable actions have to be taken. In both these situations all processors have to be started. The third situation requires just one processor to be booted after live-insertion - this is needed to support non-stop operation of the machine.

The Mach bootstrap is split into a major part done by the *master-CPU*, the processor which first got the *start_lock*, and minor parts done by all other processors, which are blocked by that lock. When the *master-CPU* has set up the system, it calls *start_other_cpus()* to free the *start_lock*. The others, knowing that at least one processor is already running, do their hardware-dependent startup, change a few kernel variables, such as *machine_slot*, and just start up a thread to get Mach running. In parallel, the *master-CPU* calls the server bootstrap, for example, that for the UNIX server.

Four distinct issues need resolution for the booting to support the recovery protocol. First the master-slave relationship of the processors during booting (and during normal operation) leads to difficulties if the *master-CPU* ever fails. Secondly there is the issue of the first point of commit during a *cold boot*. Thirdly, provision must be made for a *warm boot*. Finally, live-insertion must be supported.

7.8.2.1 Master-slave relationship

Master-slave relationships are not generally acceptable in a fault-tolerant system, since the system usually fails when the master does. Mach uses its *master-CPU* to do work that can only be done by exactly one processor. In normal operation this work reduces to keeping the time-of-day accurate, so if the *master-CPU* fails, all that is required is some forward error recovery actions to assign another processor to this work. During booting, however, all initializations are done by this processor, and a failure while it holds *start_lock* would lock up the entire machine. There are two ways to deal with this:

(a) If the machine does not start, the user can try restarting again (maybe some indication of which processor failed and hence needed removal would help). This solution is not really acceptable, since it precludes unattended rebooting.

(b) The startup can be controlled by a simple timeout mechanism such that if a processor fails during booting, other processors will be alerted by expiry of the timeout. Since Mach is not yet running, the timeout mechanism has to be implemented using additional locks and counting variables. Each processor P_i which does not get the *start_lock* can attempt to get the previous additional lock L_{i-1} (of which there should be at least as many as there are slave processors) while counting down from a huge number. If a processor gets the previous additional lock L_{i-1}, it unlocks its own additional lock L_i so that the processor holding the next additional lock L_{i+1} can continue at this level. If it times out, the previous (possibly *master-CPU*) processor P_{i-1} has probably failed, and so it unlocks its own additional lock L_i and starts at the new level again, counting or running the boot code as if it just got the previous lock. Therefore no processor failure can prevent another processor from running the boot code. The synchronization scheme is shown in Figure 7.22.

Although (b) above is a simple mechanism that combines error detection with error recovery, it suffers from one problem: the *master-CPU* might already have changed some data items during the different phases of system initialization, and so the initial values will have to be restored somehow. For example:

(a) The relevant initial values of the kernel data could be stored as a different data set, which could be copied into the relevant data structures at the beginning of the bootstrap. This has the disadvantage of increasing the size of the kernel data, quite apart from the difficulty of establishing just what needs to be copied a priori.

(b) When the kernel image is being loaded, it could be loaded into both banks of SM, and then *bank2* could be copied to *bank1* at the beginning of the bootstrap. The problem here is that the image loading would then overwrite any checkpoints in *bank2*, thereby precluding *warm boots*.

(c) The system could be *RESET* whenever the *master-CPU* failed during startup. This requires that the whole system can be *RESET* by software, and that this does not cause the failed processor to restart. This is what we assumed in the design of the SM and DPU, and is the basis of the proposed boot handler.

7.8.2.2 Cold boot

Assuming that the above , the original question arises: ”what is defined as the start of time, what happens at the start of time, when is the first checkpoint taken ?” The kernel has to be in a consistent state to which it can return if a rollback arises, and this should not require special precautions. The first suitable state is reached when the *master-CPU* is ready to `start_other_cpus()`. At this point all the kernel data structures are initialized, all the component parts of Mach are running, the *master-CPU* is about to start up a server and the slave processors are

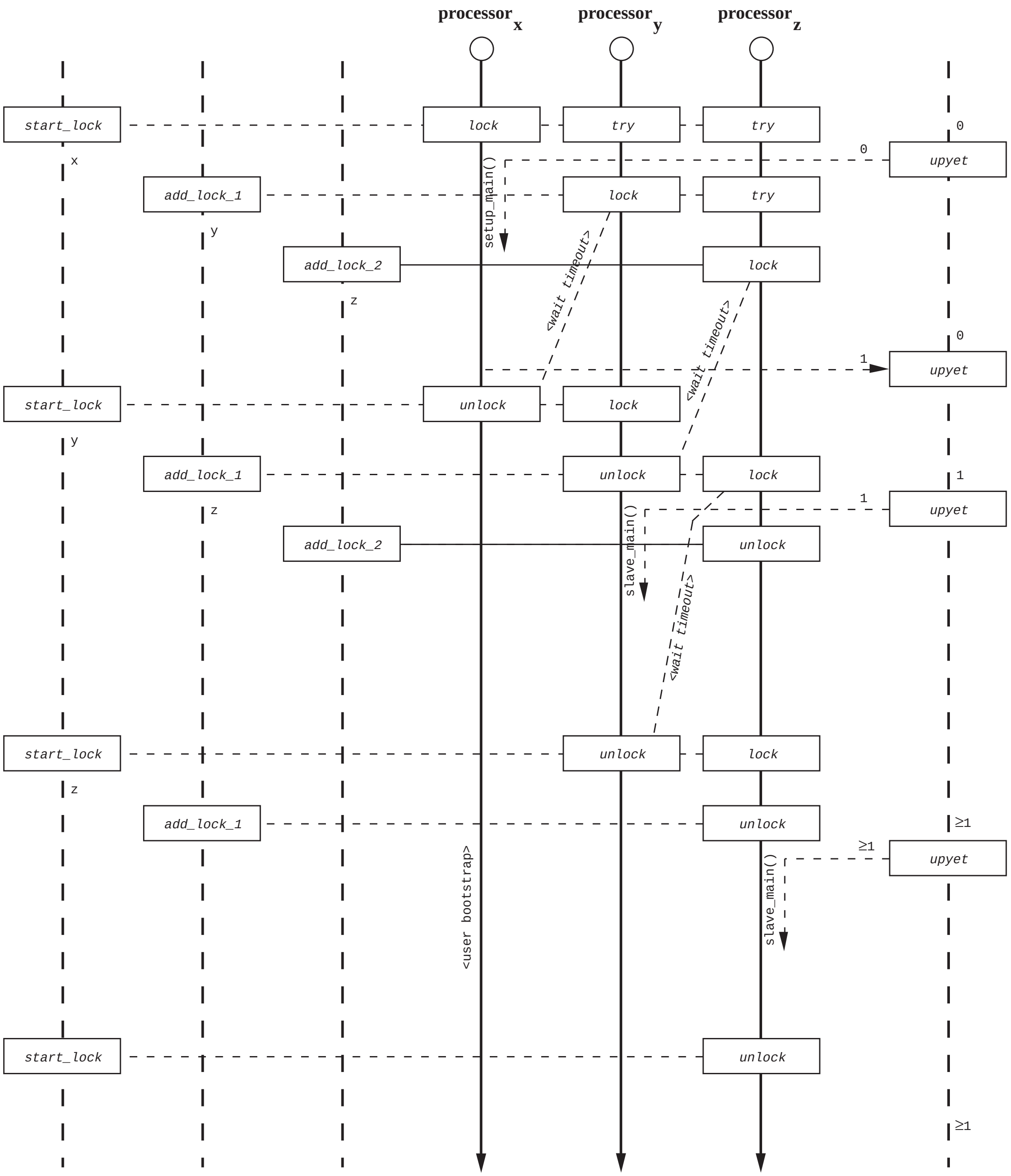


Figure 7.22: Proposed master-slave synchronization

waiting for the *start_lock* or an additional lock. If the *master-CPU* subsequently fails, then a slave processor can become the master and can continue with the server startup. Therefore it is here that the first checkpoint is taken by the *master-CPU*, and for a *cold start* this defines the start of time.

As long as no slave processor is running, the *master-CPU's* first checkpoint will not be used in a rollback, and so all processors have to start error detection and checkpointing immediately after they start running. This is done by activating a Mach timeout for each processor. For the *master-CPU* this is done automatically by the first commit, whereas the slave processors activate their timeout just before they choose their first thread (they do not need to take a checkpoint at this time because everything that is needed is contained within the *master-CPU's* first checkpoint). Any failure of a slave processor can be ignored at this stage.

There is a critical region during the startup of the slave processors, where `slave_main()` first indicates that it is alive and well, and then starts its checkpoint timeout - no checkpoint interrupt can be allowed to occur during the execution of this sequence of code, otherwise different processors might have different values for the number of processors and their state.

7.8.2.3 Warm boot

A *warm boot* is very similar to a *cold boot*. Up to the moment where the *master-CPU* takes the first checkpoint, exactly the same will happen, but in the case of a *warm boot* the *master-CPU* cannot take a checkpoint because this would destroy the last checkpoint before the system stopped. Instead, the *master-CPU* can roll back the old checkpoint from the stable memory and then do the usual forward error recovery actions to determine the number of available processors, etc.. This has no effect on the other processors, which will choose the first thread as in the case of a *cold boot*, except that in this case it will be unlikely that the first thread will be the *idle_thread*.

How does the system know that this is a *cold boot* ? One solution requires a flag to be set in an area of memory that is not recovered during recovery (i.e. in a non-recoverable memory, see Section 7.1.3). This flag can be set after a checkpoint, reset during a regular shutdown, and examined during booting.

7.8.2.4 Live insertion

The situation where a processor boots while the system is running is exactly the same as that of a non-*master-CPU*; it tries to get the *start_lock* or an additional lock with timeout, and does the normal slave startup. No further provisions are needed.

7.9 Proposal 2 : Non-standard I/O process handling[12]

Lest we become complacent, let us now examine a second proposal: a potential solution to the more difficult problem of non-standard I/O processes. We have proposed above that processes may be classified in two different types depending on the recoverability of these processes operations. A process which only executes internal operations (computation, memory accesses) is called a standard process, whilst a process which deals with external operations, such as I/O, is called a non-standard process [Banâtre et al 92a]. A standard process state is recoverable and automatically managed by the recovery protocol, but a non-standard process isn't. Hence problems may arise with non-standard process handling, mainly I/O loss or duplication. No general solution exists to solve these problems, because non-standard process handling is highly dependent on machine-specific aspects.

7.9.1 I/O handling in Sequoia

By way of example, let us present the main principles of the I/O handling of the Sequoia computer [Bernstein 88]. This is a tightly coupled multiprocessor: each processor has direct access to all memory and I/O resources and communicates with other processors through the shared memory. It presents a hardware approach to fault tolerance. A Sequoia computer consists of processor elements (PE), memory elements (ME) and I/O elements (IOE) connected by a system bus.

The system may experience a hardware fault (in a PE, ME, IOE...) at any time. When a fault occurs, it must be possible to recover the process that experienced the fault without losing its process state, or losing or duplicating I/O operations. So as to recover that state, a special mechanism is used: all writable pages are shadowed on two memory elements. These copies are named the backup copy and the primary copy. When a processor element flushes its cache, it actually flushes twice. First, it flushes to the backup copy of its page and, when that is complete, it flushes to the primary. There are two interesting aspects for I/O handling:

(a) Avoiding lost and duplicated I/O : Each I/O element has a queue in main memory containing pending I/O operations to that element. To perform an I/O, a processor element constructs a description of the operation in its cache and then flushes that description to the appropriate queue in main memory. Once an I/O is successfully appended to that queue, it is guaranteed to be performed. An I/O is deleted from

[12] This section contributed by Marylène Clatin and Christine Morin, IRISA/INRIA, Campus universitaire de Beaulieu, F-35042 Rennes cedex France.

the queue only after the I/O element acknowledges its completion. The flush mechanism ensures the I/O is not lost and the process will not repeat it. Moreover, the I/O element maintains a list (in its local memory) of all in-progress I/O operations. Indeed, after a fault, it is not always possible for a processor element to tell, without assistance, which I/O operations have actually been sent to the I/O element. This might be the case, for example, when a processor element fails while sending operations to an I/O element. Therefore, in order to avoid losing I/O, each I/O element is interrogated after a fault to determine which of its queued I/O operations in main memory it has accepted (i.e. which are present in its list in local memory).

(b) I/O failures : Disk failures are handled using dual-ported mirrored disks on different I/O elements. The kernel routes each write to both disks of a mirrored pair. It balances the read load by sending half to each disk. If a disk fails, the other disk picks up the full load.

In the following, we present a potential solution which has several common aspects with this one, although realized in a different manner.

7.9.2 Reliable I/O Design Principles

For the FASST architecture, a cache is associated with each processor, and a shared memory is composed of one or more stable memory units. A recovery protocol determines the set of process states which together constitute a consistent state of the system. Processes communicate through shared memory, thereby creating dependencies. Processes establish recovery points, and when a fault occurs all dependent processes are rolled back together to their last recovery point state, so that they can restart their execution. To establish the recovery point (for the set of dependent processors) a call is made to the SM function `commit_group`.

Again by way of example, let us assume that this architecture is implemented on a Corollary machine [Corollary 92, Ferrara 91], as in Figure 7.23. P1, P2, P3, P4 and $\mathrm{P}_{\mathrm{I/O}}$ are computing processors. $\mathrm{P}_{\mathrm{I/O}}$ (called the I/O processor) also supports I/O requests. $\mathrm{P}_{\mathrm{I/O}}$ is the only processor which has access to the AT bus so as to manage devices (their later models employ symmetric processors to overcome this restriction). All processors access shared memory through the Cbus. Most of the shared memory of the Corollary machine must be replaced by stable memory to allow implementation of the recovery protocol, but, as outlined in Section 7.1.3, some standard shared memory is still needed to handle unrecoverable objects (such as I/O).

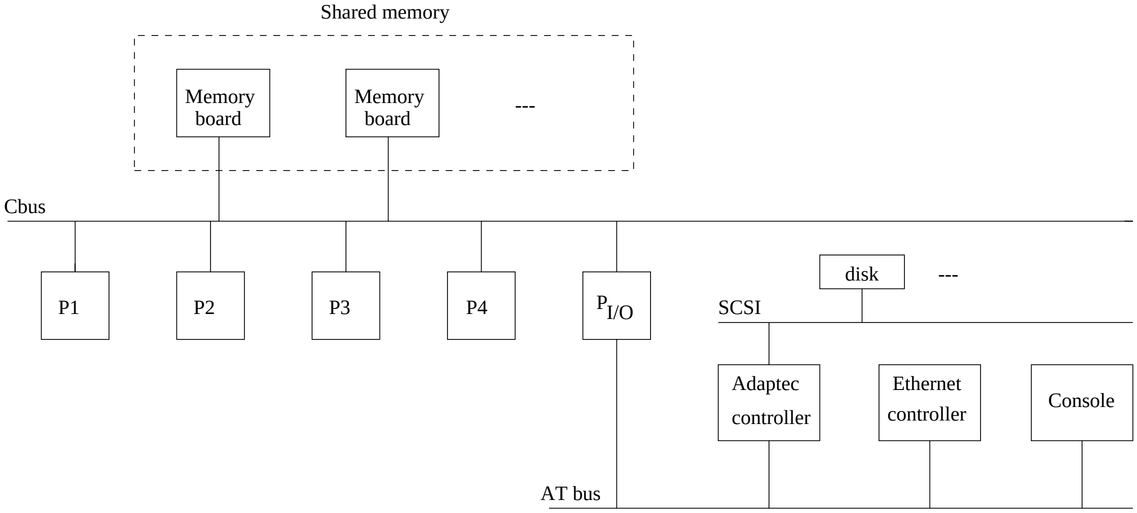


Figure 7.23: Corollary architecture

Our aim in this proposal is to ensure, in case of rollback, the coherence of the system when it includes threads that issue I/O operations, and in particular to avoid both loss and duplication of I/O operations. When a rollback arises, a standard thread is rolled back to its previous recovery point, so it will restart its execution in a coherent state. The case is different for a non-standard thread: when a rollback arises, it must not forget non-idempotent operations, for example, it must not forget if it has initiated an I/O request. Despite its particular behaviour, the non-standard thread state must be coherent with the system.

We consider the following assumptions:

(a) the I/O processor is reliable,

(b) communications between threads are reliable,

(c) devices (disk, console, ...) are failure-free.

Hypothesis 1 *We are only dealing with the failure of a computing-only processor, supporting a thread involved in an I/O operation.*

7.9.2.1 Model of I/O handling

Let us consider the thread which requests an I/O instruction and the thread which realizes the I/O operation itself as two different entities. When a standard thread *t* needs an I/O operation, it sends a request to the device port associated with the device. Then, the non-standard thread *tio* receives this request from the port, realizes the I/O operation on the device and acknowledges the I/O by sending a result message to the reply port associated with the request port. Figure 7.24 illustrates these communications.

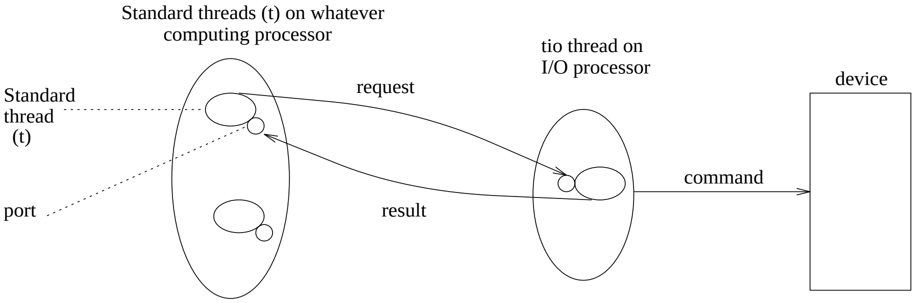

Standard threads (t) on whatever computing processor
tio thread on I/O processor
device
Standard thread (t)
request
port
result
command

Figure 7.24: Communications in an I/O operation

An I/O operation happens in the following way:

(1) *t* sends a request to the port associated with the device,

(2) *tio* receives the request from the port,

(3) *tio* executes the I/O request,

(4) when the I/O is completed on the device, *tio* sends the I/O result to the reply port associated with the request port,

(5) *t* receives the result from the reply port.

The communication phase is characterized by message sending and receiving via ports. Therefore, let us consider a protocol which applies at this level, using atomic operations to introduce commits into message sending or receiving, specifically in order to prevent the system from losing or duplicating I/O operations due to a processor rollback (this is further justified in [Jöhnk et al 92]).

7.9.2.2 Proposed general I/O protocol

Before presenting the protocol detail, let us specify its main features. An atomic operation will be denoted in one of the two following ways:

```
(1)  begin atomic
          sequence of code
      commit_group

(2)  begin atomic
          sequence of code
      commit(zone)
```

These atomic operations have the two following properties. Either the code of the atomic operation will be fully executed, either it won't be executed at all. This is the all or nothing property. Moreover, it has the indivisibility property, i.e. the intermediate states are not visible.

The first type of atomic operation will be used by the standard thread, whilst the second one will be used by the non-standard thread. These are different because a standard thread is involved in the recovery protocol and therefore is able to use its `commit_group` primitive to validate the atomic operation and trigger a commit operation for the set of dependent processors, whilst a non-standard thread is not involved in recovery protocol. Therefore, let us introduce a new primitive `commit(zone)`, which validates the SM area identified by the name `zone` and triggers a copy of the corresponding SM blocks from *bank1* (current value) to *bank2* (recovery value), thereby rendering persistent the information contained within those blocks. This function validates the second type of atomic operation, since the sequence of code only operates on the SM area `zone`.

The ports used to communicate with devices are standard ports held in stable memory. On the other hand, all data structures concerned with I/O handling on the device are held in non-recoverable memory associated with the I/O processor. This information (see Section 7.9.3) reflects the I/O state on the device, therefore it must not be lost as a result of a rollback.

Both the standard and non-standard threads are able to access device ports in stable memory. A standard thread is recoverable, whilst a non-standard thread isn't since it accesses unrecoverable objects (devices). In order to avoid dependencies created by the interactions between these two types of threads, atomic operations are used for sending and receiving messages from ports. After a non-standard thread has received the request message from the device port, it performs the I/O operation on the device (using the data structures in non-recoverable memory) and then sends the result message to the reply port associated with the device.

An I/O operation takes place synchronously as summarized in Figure 7.25. The standard thread is blocked at (1) waiting for the I/O result message. The non-standard thread sleeps at (2) waiting for a request, and is woken up when such a request arises.

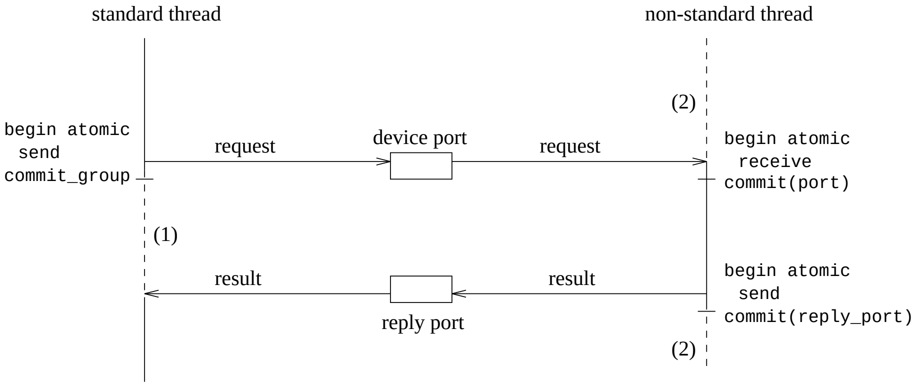


Figure 7.25: General I/O handling

The standard thread *t* sends a request to the device port and commits in an atomic operation. Then, the non-standard thread *tio* receives this request from the port and validates the new port state in an atomic operation too. *tio* performs the I/O operation on the device. At completion time, it sends the result message to the reply port associated with the device, and validates the port state again in an atomic operation.

This protocol is justified below.

7.9.2.3 Sending of the I/O request

If we assume that a commit is not performed when the standard thread *t* makes a request for an I/O operation, then the following situation may happen: if we suppose *t* has committed at any time before it sends its request, a processor failure will cause *t* to rollback and then it will again send its request to the device port. But, during the interval between the message sending and the rollback, the first request may have been serviced by *tio*. Hence, the request will be duplicated.

On the other hand, if the thread sends its request and saves a recovery point in an atomic operation, we avoid any duplication in the case of a rollback. The message sending delivers a request to the device port, but *tio* won't receive it until *t* has validated it by a commit (thanks to the indivisibility property of the atomic operation). Therefore, *t* cannot resend this same request in the case of a rollback. This also ensures coherency between the standard and non-standard threads: *tio* can only execute an I/O operation for which the request has been validated by *t*, thereby avoiding an orphan I/O execution in the case of a rollback.

The standard thread uses the following atomic operation for sending its request:

```
begin atomic
   send request to the device port
commit_group
```

7.9.2.4 Receipt of the I/O request

For *tio*, things are different because it is not involved in the recovery protocol since it is not recoverable. The non-standard thread *tio* will receive a request message from the device port in stable memory, and then will perform the I/O on the device using I/O handling data structures in non-recoverable memory. If no commit is performed, the following situation can arise: if the SM is rolled back after *tio* has received the message from the port and possibly started its treatment, then the SM will recover a state where the message is still present in the port since ports are held in stable memory. Hence, *tio* will be able to receive it again, and again execute the treatment, and the I/O operation will have been duplicated.

The solution consists of receiving the request from the port and validating the new port state in an atomic operation. When *tio* accesses the port in stable memory, it must neither create any dependency with other threads, nor validate any port modification performed by another thread. This is achieved through the atomic operation, more precisely thanks to its indivisibility property, thereby avoiding any duplication and dependency problems.

```
begin atomic
   receive request from the port
commit(port)
```

7.9.2.5 Sending of the I/O result

Symmetrically, problems may arise in sending the result from the non-standard thread. We can be faced with a situation in which *tio* sends a result message to the reply port, and then the SM is rolled back to a state where the result message is no longer present in the reply port. The result message will have been lost, and *tio* won't resend it since this thread is not recoverable. Sending the result in an atomic operation will solve these problems:

```
begin atomic
   send result to the reply port
commit(reply_port)
```

We underline that for the receipt of the I/O request and the sending of the I/O result, the non-standard thread is committed so as to validate the port states in the SM; these commits will have no effect on the device data structures stored in non-recoverable memory or the non-standard thread execution code. We should also note that no commit is necessary when the standard thread receives the result message from the reply port; if a rollback arises after the non-standard thread has sent the result message to the reply port and committed, then the standard thread will find it again (remember ports are held in the SM).

Moreover, the reply port validation `commit(reply_port)` after sending the result won't create any problem by somehow validating an incorrect port state, since *t* is the only thread which has a *receive* right for this reply port, and since *t* is blocked waiting for the result message. Such a problem could arise if several *receive* rights existed for the same reply port or if asynchronous I/O was utilized, essentially as a result of not committing when the standard thread receives the result message from the reply port.

The atomic operations are the basis of the protocol. Indeed, the indivisibility property ensures the modifications performed on ports are visible only after the end of the atomic operation, while the all or nothing property ensures both no I/O loss nor duplication arises due to a rollback operation. From a general point of view, however, we should note that no `commit_group` operation should arise while performing a `commit(zone)` because they both operate on the SM - a simple solution consists of delaying the `commit_group` while the `commit(zone)` operation is running, since the latter is much faster.

7.9.2.6 The I/O processor

In the Corollary machine the I/O processor is also a computing processor. It will support both standard and non-standard tasks, and therefore standard and non-standard threads. However, we will have to separate clearly standard and non-standard threads, because the former are recoverable whilst the latter aren't, and therefore any dependencies created between these two types of threads will cause difficulties. A processor rollback must never induce rollback of a non-standard thread because of a dependency created by an access to a shared data structure, such as the Mach run queue.

7.9.3 Device management in the Mach microkernel

This section is inspired by the section ”Device management” of [Jöhnk et al 92]. Devices are represented as device ports, and communication with a device in Mach is performed through Inter-Process Communication (IPC). The microkernel expects each device to provide a small number of functions, and it converts messages to the device ports into a call to one of these functions. These functions provide a uniform call interface; differences between devices are introduced through interpretation of the parameters:

(a) `device_open(...)`

(b) `device_close(...)`

(c) `device_read(...)`

(d) `device_write(...)`

(e) `device_get_status(...)`

(f) `device_set_status(...)`

(g) `device_map(...)`

All data structures associated with devices will be handled in non-recoverable memory, since devices are unrecoverable objects. Indeed, if these data structures were handled in recoverable memory, we would be faced with coherence problems. For example, consider a device read message: the non-standard thread updates the kernel data structures concerned and initiates a read operation on the device; if the thread is rolled back at this time, then the data structures will be recovered with their backup value, but the read operation will have been initiated on the device. Hence, we will be faced with a coherence problem between the kernel data structures representing the device state, and the real device state. The solution consists of handling those data structures in non-recoverable memory.

The microkernel is structured such that all devices share a generic layer called the device service. Another layer contains the specific code which depends on the hardware device. Specific layer data structures depend on the device type. Hence, let us first look at the generic layer, then at the specific layer for two different device types, SCSI disk and console, and their associated data structures, organization principles and an example *device_read* or *device_write* function.

7.9.3.1 Generic layer

All the data structures associated with this layer are illustrated in Figure 7.26. All the devices are represented by ports. To obtain the device port for a particular device, a *device_open* message is sent to the *device_master_port*. Only privileged tasks have *send* rights for the *device_master_port*. A string is sent within the message which names the device in a system specific manner: this name is composed of a major device number and a minor device number. The kernel interprets this name and extracts the major and the minor device numbers.

The major device number is an entry in the *dev_name_list* table that permits the device type to be determined. One of the entries in this table is *dev_ops*; it contains pointers to the code for each function of the interface. If the device is currently in use (i.e. it has been opened), a pointer to the *device* structure representing the device will be found in the *dev_number_hash_table*. This table is hashed on minor numbers. If the device is not opened, a *device* structure is created which represents the device and contains, among other things, some state information, a pointer to the device specific *dev_ops* and a new port. This port is saved into the field *port* within the *device* structure and is also returned to the task opening the device; other tasks which will open the device will acquire a *send* right

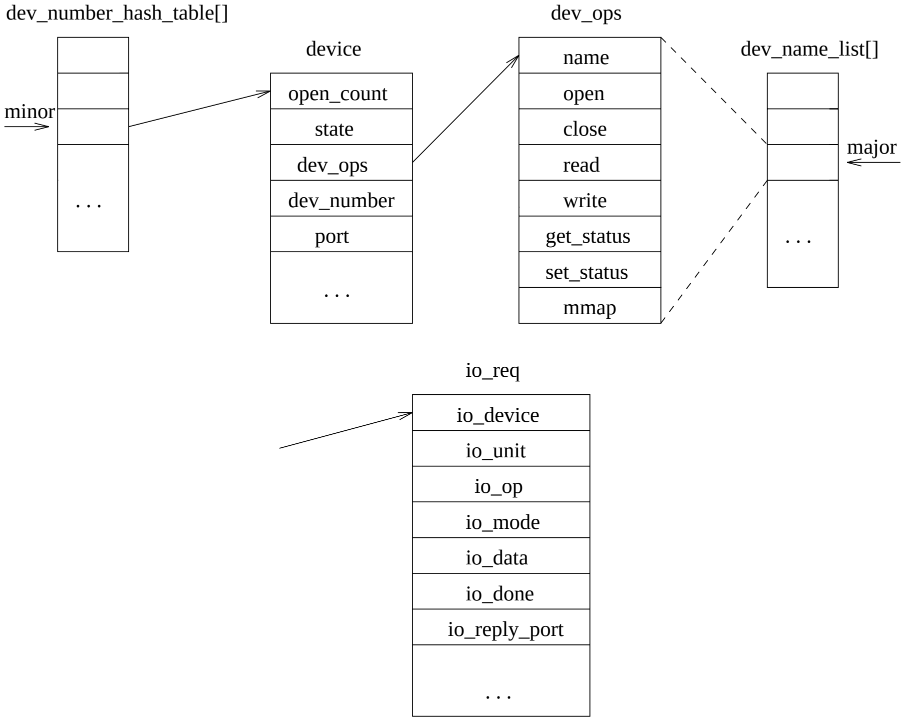


Figure 7.26: Generic layer data structures

for this same port. This port is then used to communicate with the device. Several functions are defined in the interface, but the device specific driver only needs to support those functions which are appropriate for its device.

When a function to open, to read or write data to the device is invoked, the generic code creates an I/O request structure (*io_req*) which all device specific code understands; it contains information about the requested operation type, the mode and the device to use. The structure will be passed to the device driver code for the operation. Such a structure is only created for operations that can be delayed (i.e. open, read or write). Two entries in the structure specify the return path from the driver code to the user level: a finish-up routine (*io_done*), which generates the message to be delivered to the reply port and sends it, and a port on which a reply is to be sent (*io_reply_port*).

7.9.3.2 SCSI device specific layer

The SCSI device has three layers. The data structures concerned are illustrated in Figure 7.27. The top layer handles different SCSI devices (disks, tapes, ...). Different classes of devices are handled in various ways, for example, tapes are handled somewhat differently to disks, even for the same operation. These differences are encoded in an entry in the *scsi_devsw* array. Entries in this array provide yet another generic interface for reading, writing and so on. Such an entry has the *scsi_devsw_t* structure, and provides pointers to the functions for the specific class.

The middle layer encodes SCSI commands. A SCSI controller (a *host adapter*) handles at most 8 physical device units which are referred to as *targets*. Each controller is associated with an index into the *scsi_softc*[] table, which contains pointers to *scsi_softc_t* structures. A *scsi_softc_t* structure contains some general information for the controller and pointers to per-target status information (*target_info_t*), which for each target points to a list of outstanding I/O requests (*ior*), and also to the *scsi_devsw* entry, amongst other things.

The bottom layer supports few commands. The kernel relies on a single call (*aha_go*) to initiate a command. Interrupt service routines maintain the transfer, and will indicate termination to higher levels. Some of the data structures of this layer are illustrated in Figure 7.28. An *aha_softc* array provides pointers to *aha_softc_t* structures which represents state descriptors (I/O port, number of targets alive on this SCSI bus, pointer to *scsi_softc_t* structure, ...). Other arrays (*aha_minfo* and *aha_dinfo*) provide information for each device controller (*aha_minfo*) and each device unit (*aha_dinfo*).

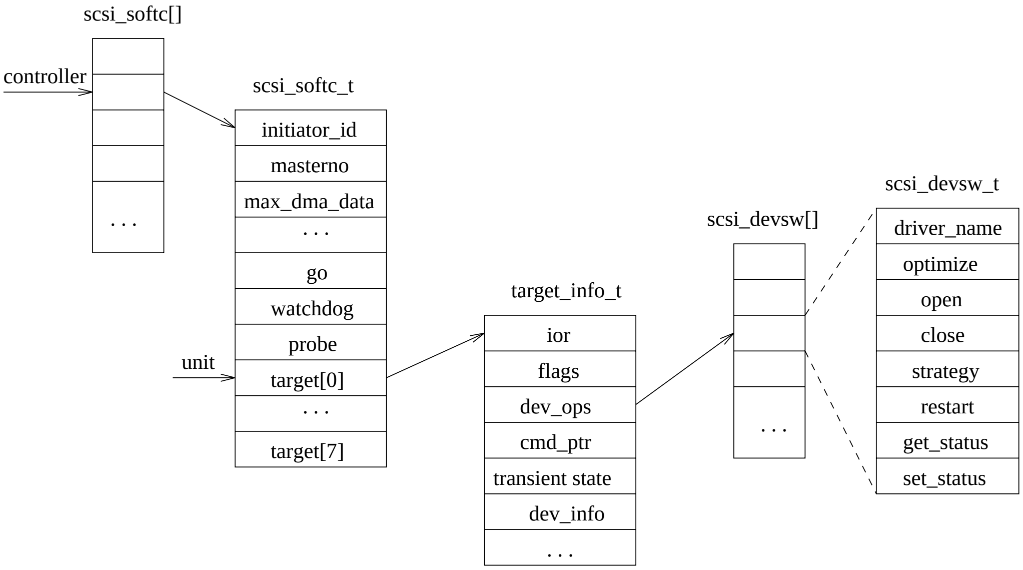


Figure 7.27: SCSI device specific layer data structures

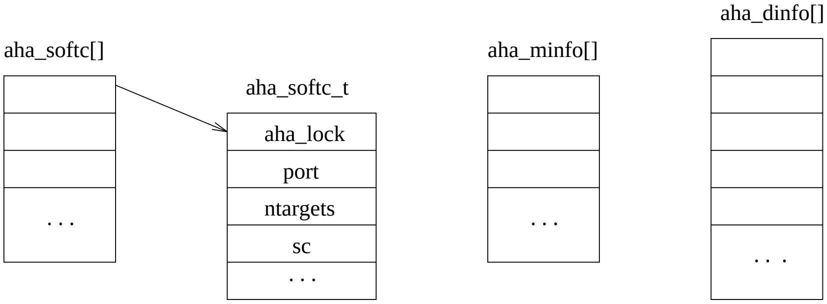


Figure 7.28: SCSI device disk specific bottom layer data structures

As an example, the SCSI device disk read path is shown in Figure 7.29. A *device_read* message is sent to the device port by a thread. The kernel receives this message and passes it to the generic device handling code for read (*ds_device_read*). This code creates a device read I/O request (*io_req*), and then invokes the *dev_ops* read function (*rz_read*), passing along the I/O request.

At this point the thread has reached the specific layer, i.e. the top of the SCSI device driver. The minor device number is used to determine the controller (the index for the *scsi_softc* array), the device unit and the partition. Then the device unit is used to determine the target in the *scsi_softc_t* structure. The I/O request is linked into the *target_info_t* request (*ior*) chain, using a *scdisk_strategy* and then a *disksort* function. These strategy routines sort requests to minimize head movement. Finally, we find a call to the *scdisk_start* function for the request which has reached the head of the queue. This function is both the start and completion routine for disks; it calls the *scdisk_start_rw* function which then invokes the *scdisk_read* routine of the driver middle layer.

This *scdisk_read* function builds an I/O command structure (*cmd_ptr* of *target_info_t*) and then invokes the *scsi_go* routine. The latter will just call the machine specific function to start the I/O, and invoke the bottom layer of the driver. There, the command is passed to the *aha_go* function, which locks the thread onto the I/O processor, before initiating the operation on the device; if the thread is not on the I/O processor, it has first to ask for migration to that processor, and then blocks (the scheduler will reactivate it later).

Only the I/O processor can receive device interrupts. When an interrupt arises, it is passed to a device specific routine *aha_intr*. At the end of the I/O operation, the thread is unlocked (and reverts to its original processor, if it wasn't the I/O processor), and the target specific restart function *scdisk_start* is invoked. Then, the *iodone* routine is called which deals with delayed reply sending. It invokes the *io_done* function of *io_req*, i.e. *ds_read_done*,

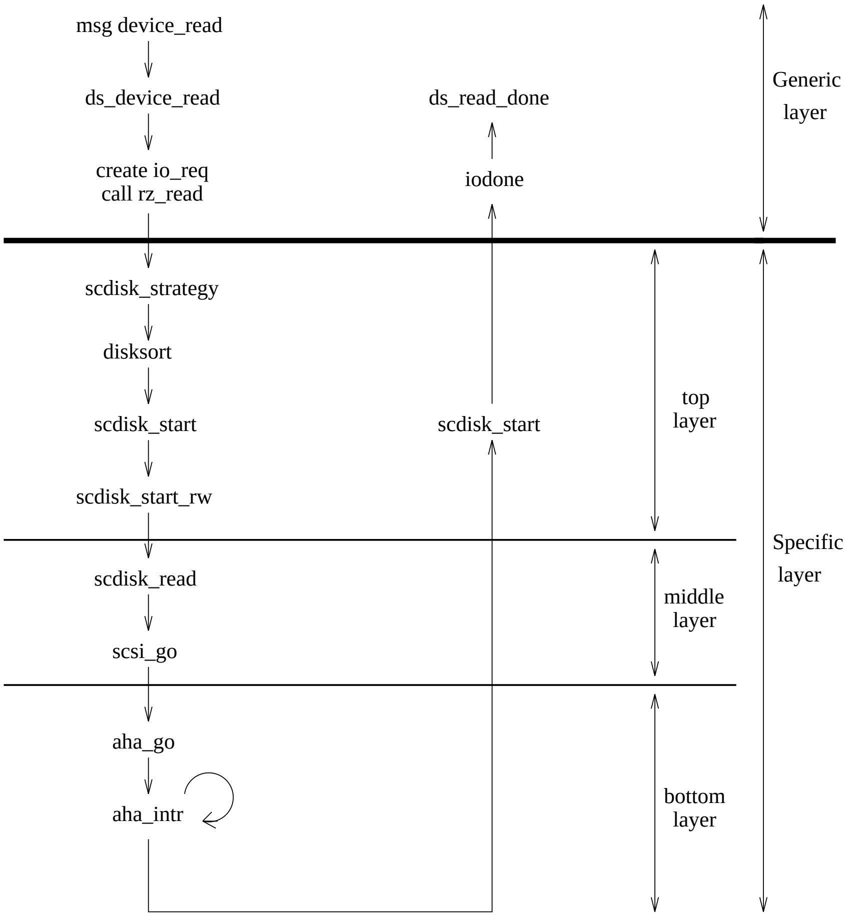


Figure 7.29: SCSI device disk read path

which replies to the message sent to the device.

7.9.3.3 Console device specific layer

The specific layer is simplest in the case of the console device. The data structures are shown in Figure 7.30. A *kd_tty* structure is associated with the console device. This structure contains pointers to an *input queue*, an *output queue*, a *kdstart* function, a *kdstop* function, and so on. The input and output queues are circular buffers which store characters. Different functions are also provided at this level (*kd_dput*, *kd_dmvup*, *kd_dreset*, etc.) which define the interface for the device specific layer.

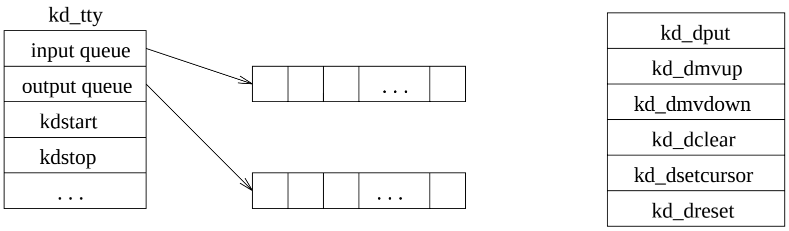


Figure 7.30: Console device specific layer data structures

As an example, the console device write path is shown in Figure 7.31. The generic layer treatment is similar to that for the SCSI device, except for the functions called. A thread sends a *device_write* message to the device port.

The kernel receives this message and passes it to the generic device handling code for write (*ds_device_write*). A device write I/O request (*io_req*) is created. The *dev_ops* write function (*kdwrite*) is invoked, passing along the I/O request.

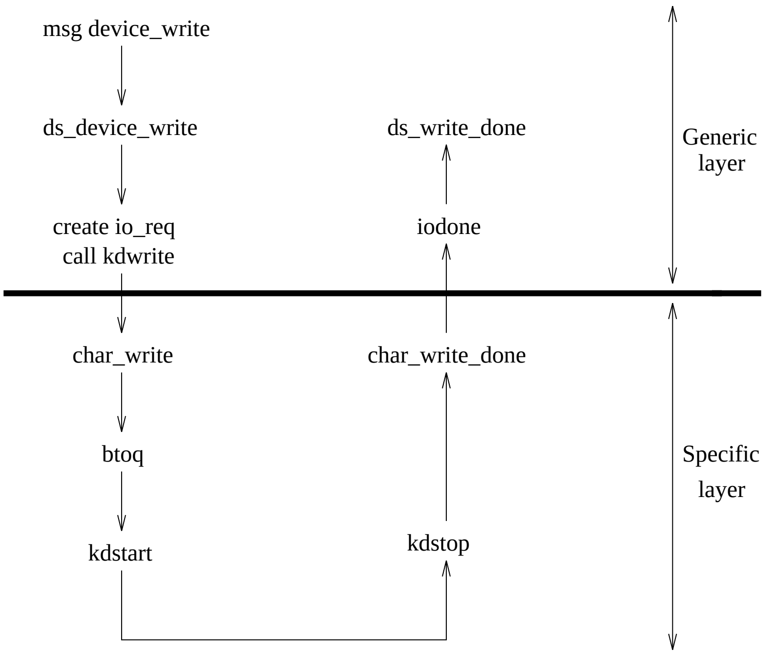


Figure 7.31: Console device write path

At this point the thread has reached the specific layer, i.e. the console driver. The *char_write* function is invoked, which calls the *btoq* ("buffer to queue") function. This last function copies *io_req* data (*io_data*) to the *kd_tty* console output queue. The transfer is then performed by calling the *kdstart* function. If the thread was running on the I/O processor, it is locked on that processor; otherwise, it first asks for migration onto the I/O processor and blocks (the scheduler will reactivate it later). This is achieved in the *kdstart* routine, before writing all the characters to the console in a loop. The thread is then unlocked and reverts to its original processor. The *kdstop* function begins the return path for the device reply. The *char_write_done* routine is invoked for the delayed reply when the output queue has been emptied. Then, the *iodone* routine invokes the *io_done* function of *io_req*, i.e. *ds_write_done*.

As a further example, let's look at the console device read path (see Figure 7.32). In a similar fashion to the write path, when a thread asks for a *device_read* message on the console, the following steps take place in the generic layer: send a *device_read* message on the device port, call the *ds_device_read* function, create an *io_req* data structure, and call the *kdread* function.

Again, at this point the thread has reached the specific layer, i.e. the console driver. The *char_read* function is invoked which delays the treatment until the user has entered the number of characters specified in the request. The console device read path is a little bit different at the bottom of the specific layer, in that an interrupt service routine reads in the characters from the keyboard and puts them into the *kd_tty* input queue (using a *ttyinput* function). Console interrupts are only used for input. Thus the input queue is filled up in the interrupt service routine, somewhat independently from any function of the console specific layer read path. We should note that, because of this independence, the thread need not be migrated to nor locked onto the I/O processor. The I/O processor fills in the input queue on interrupts, and then the thread call to *qtob* function just copies the characters from this input queue to the *io_data* field of *io_req*. Here again, *iodone* will finally call the *io_done* function of the *io_req* structure, i.e. *ds_read_done*.

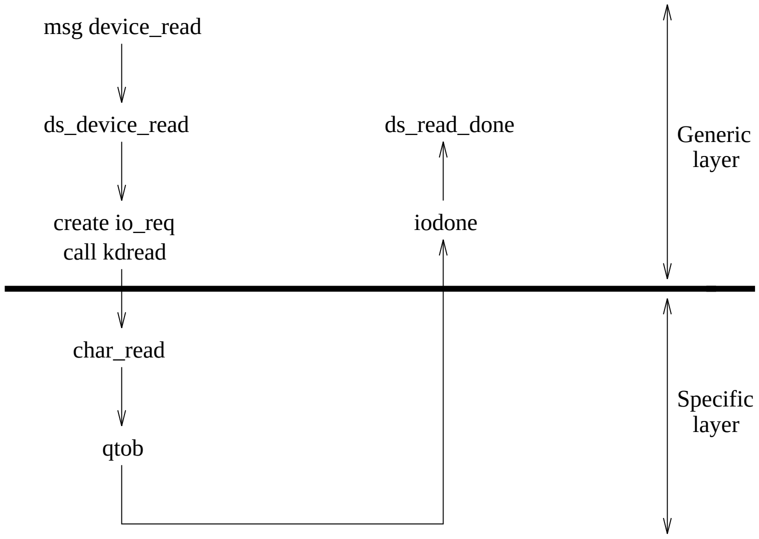


Figure 7.32: Console device read path

7.9.4 A first implementation proposal for reliable I/O

Clearly, Mach I/O handling is different to the vision of standard and non-standard threads presented in Section 7.9.2. In fact, there is no distinction between threads. A thread which requests an I/O operation can be supported by any one of the Corollary computing processors, or by its I/O processor (which is also a computing processor). If the thread is located on the I/O processor, it is locked there; otherwise it first has to migrate to the I/O processor. Therefore, we have to adapt our model for Mach, and vice versa.

The first problem concerns the I/O processor. As we have seen in Section 7.9.2.6, standard and non-standard threads should be clearly separated on this processor in order not to create dependencies between them. One solution is to separate the run queues on the I/O processor: one for standard threads, and another one for non-standard threads. A simpler solution is to prevent standard threads from running on the I/O processor, i.e. only allow non-standard threads to run on that processor, using its existing run queue, while the standard threads run on the computing-only processors.

The second problem is how to handle device data structures. The most obvious approach is to try to handle the generic and specific layers data structures in the SM, and to handle the bottom layer data structures in non-recoverable memory (*NRM*). The main problem with such a solution concerns dependencies created by threads running on the I/O processor, which access the SM. We have seen non-standard threads are not recoverable, so many forward error recovery actions (FERA) would be needed in the case of rollback to update the SM state, so that it would reflect the device state. Indeed, not only the bottom layer data structures are involved, but also some of the generic layer (like the *io_req* structure which is built each time a write, read or open operation is requested) and some of the specific layer (for example, the *ior* field of the SCSI device's *target_info_t*, which points to the delayed operations chain). However, we can handle the generic layer *dev_name_list[]*, and any specific layer structure that only contains pointers to function code (such as the SCSI device's *scsi_devsw[]* table) within the SM, since these are not modified (and hence no dependency is created by accesses to them).

Holding the generic layer *device* structure or the *dev_number_hash_table[]* in the SM is also possible if we consider that only *device_write* and *device_read* requests are critical (because they access the physical device), and that *device_get_status*, *device_open*, etc., can be re-executed. However, such a solution can create coherence problems between SM and *NRM* states: for example, an *io_req* structure may contain in its *io_device* field a pointer to a *device* structure that is no longer valid because of a SM rollback. Again, forward error recovery actions are necessary (mainly concerning pointers and *device* structure references) in order to build a consistent state. Moreover, considering *device_write* and *device_read* requests as the only critical operations may not be such a good idea, since a *device_open* request can also be delayed like a *device_write* or a *device_read*, but does not access the physical device. Introducing a classification of request types is not so straightforward as it might seem.

On the other hand, the disk specific layer data structures *scsi_softc[]*, *scsi_softc_t*, *target_info_t* contain state information, pointers to *io_req*, and so on, representing the device state, and therefore, holding them in *NRM* is the better solution that avoids many forward error recovery actions. Similarly, the console specific layer data structures for input and output queues should be held in *NRM* in order not to lose any characters.

Thus some device data structures cannot be held in the SM, whilst others can. For the latter, forward error recovery actions are often needed so as to re-build a consistent state of the system; many of these actions may be necessary after a rollback, and make it more or less difficult to handle, according to the type of memory chosen to store each data structure. Therefore, in the following, all the device data structures are stored in *NRM*. This solution has two main advantages: no forward error recovery action is needed for those structures, and I/O operations are clearly separated from the rest of the system according to their non-recoverability.

7.9.4.1 Principles

Let us assume the computing-only processors are able to access the SM, and the I/O processor has non-recoverable memory *NRM* for handling I/O data structures, where this *NRM* memory is also accessible from the other processors. Let us further assume that the I/O processor is exclusively dedicated to I/O operations on devices, and only uses the *NRM* (it does not access the SM, so as not to create any dependency). These two principles and the associated mechanisms allow a firewall to be constructed between I/O operations and the rest of the system.

Now let us consider a *standard thread* (ST) running on any computing-only processor, which requests an I/O operation. This standard thread creates another thread by a thread fork. We call this latter a *non-standard thread* (NST) because it will migrate to the I/O processor, and will perform the I/O on the device. A thread fork is used because a standard thread cannot be used for executing the request on the I/O processor. Indeed, if a rollback arose on the original processor while a standard thread was bound to the I/O processor, then the rollback would restore the run queue state at its last recovery point, and would try to restart the standard thread execution on the original processor, even though it would still be running on the I/O processor, and yet we *must* rollback a standard thread because of its dependencies with other standard threads. In this proposal, the case for the non-standard thread is different because it only has dependencies with its parent standard thread, and not with any other standard thread. Judiciously set commit operations can take care of these dependencies between the standard parent thread and its non-standard children.

The non-standard thread is able to access stable memory (and therefore initiate commits, which is the interesting point for us), as well as the I/O processor *NRM*. In order to keep as much as possible of the original I/O handling of Mach, it must also be able to locate device ports in the *NRM*.

After its creation, this non-standard thread first runs on a computing-only processor. It receives the request message from the device port in the SM, and triggers its transfer to *NRM*. Then, the non-standard thread is locked on the I/O processor and performs the requested I/O operation. At completion, it returns to its original processor, transfers the result message from *NRM* to SM, and then sends it to the reply port. The non-standard thread is then killed, and the original (i.e. the parent) standard thread can then receive the result message.

Two other interesting points must be borne in mind. Firstly, the standard thread can support rollbacks at any time between sending its request and receiving the result, including while the non-standard thread performs the I/O operation on the device. While the non-standard thread is running on the computing-only processor, it can also support rollbacks. But once it is running on the I/O processor it cannot. Thus we must be able to detect when the non-standard thread is locked on the I/O processor.

Secondly, the atomic operations presented in Section 7.9.2 are not implemented by hardware. These atomic operations have been introduced for message sending and receiving, in order to break the dependencies created by the interactions between standard and non-standard threads. If a rollback arises between the message sending (or receiving) and the commit, then the operations must be able to be restarted and the information recovered.

In the following sections, we present the steps within the proposal that characterize an I/O development:

(a) the standard thread code,

(b) the non-standard thread code executed on the computing-only processor, and

(c) the non-standard thread code executed on the I/O processor.

7.9.4.2 Algorithm for the standard thread

The standard thread on the computing processor executes the following code fragment:

```
...
send(device_port,req);            /*I/O request sending to the device port*/
commit_group();
thread_fork(ST,NST);              /*thread fork (by ST) creating NST      */
commit_group();
...
receive(reply_port,res);          /*I/O result receipt from the reply port*/
...
```

Once it has deposited its request message in the device port, the standard thread commits. This is done for two main reasons: it prevents the standard thread from returning to a state prior to its message sending, and thereby avoids duplication in the case of rollback (see Section 7.9.2.2), and it also validates the port state in stable memory. At this point, the I/O request can no longer be cancelled, and therefore can be handled by the device.

The standard thread then creates the non-standard thread by a thread fork, and commits to prevent a new fork being created as a result of a rollback. The `thread_fork` primitive does not belong to the Mach kernel interface, but it can be implemented using Mach primitives. The fork creates the non-standard thread context in the SM.

Some time later, the standard thread will receive the result message from the reply port. No commit is necessary here because the ports are held in the SM, and a commit was performed when the message was deposited in the port, and so even if a rollback occurs, the result message will still be in the reply port (again, see Section 7.9.2.2).

7.9.4.3 Algorithms for the non-standard thread running on a computing processor

The non-standard thread code executed on the computing processor is as follows:

```
receive(device_port,req);          /*I/O request receipt from the device port*/
no_io=transfer_SM_NRM(req,&no_buf);
                                   /*I/O request transfer from SM to NRM     */
commit_group();
enqueue_request(no_buf,no_io);
                                   /*request insertion in the NRM device port*/
lock_on_master();                  /*locks the NST on the I/O processor      */
       /*Here, the NST is migrated to and locked on the I/O processor.
         It will be unlocked at completion time,
         after the I/O operation has been performed.*/
res=transfer_NRM_SM(no_io);        /*result transfer from NRM to SM          */
send(reply_port,res);              /*result sending to the reply port        */
commit_group();
notify_ack(no_io);                /*NRM data structures management          */
thread_terminate(NST);             /*kills the NST                           */
```

The non-standard thread receives the request message from the device port. We should note that the port state containing the message has been validated in the SM before the non-standard thread could receive it (by the commit following the send instruction, before the fork in the standard thread code).

Then, the `transfer_SM_NRM` routine copies the request message from SM to *NRM* memory, and returns an I/O number which will be used later for finding the result message. A commit (let us call it C1) validates the new port state. The request message is then inserted in the device port in *NRM* (if it has not been done yet, that is if no rollback has occurred). Special measures have to be used to prevent any commit or rollback during enqueueing; these are explained below in the description of `enqueue_request`.

Now the non-standard thread can be migrated to and locked on the I/O processor to perform the I/O operation. It is unlocked at completion time, and returns to its original processor. Then, the `transfer_NRM_SM` routine copies the result message from *NRM* to SM, so as to send it to the reply port. A commit validates the port state in the SM. The last thing the non-standard thread has to do before terminating is to acknowledge the I/O result receipt for the *NRM* memory management.

If a rollback arises while the non-standard thread is running on the I/O processor, the run queue of the original computing-only processor is restored to its last commit point state. This commit is the one we called C1 above, which follows the request message receipt and its transfer from SM to *NRM*. Some forward error recovery actions (discussed in Section 7.9.4.5) are necessary after such a rollback in order to update the non-standard thread state in the SM, and also to update the run queue so that it reflects the fact that the non-standard thread is now running on the I/O processor. During this time, the non-standard thread on the I/O processor keeps on executing the I/O operation on the device.

Now let us look at each of the new functions invoked by the code segment above.

∗∗ Transfer_SM_NRM

In *NRM*, we have ports associated with devices, just like the ones in the SM. Transfers are needed between these two types of memories for request and result messages. It is important to remember that the transfer is performed by the non-standard thread while it is executing on a computing-only processor. The problem with such a transfer is that the computing-only processor may fail anywhere during the operation, thereby triggering a rollback. The non-standard thread would give up the transfer it was executing, and the rollback would cause the transfer to be performed again. In the end, the *NRM* would have the complete message plus a part of this message in the device port. Clearly such a situation must not happen.

To prevent this, this proposal uses buffers for the transfer. The request message is copied into a buffer which is then locked, and, when the transfer is complete, the request is inserted in the port. Timeouts must be used in order to detect a processor failure during the transfer operation, so when a buffer is allocated, a timeout is triggered. If the timeout expires, then it means that the non-standard thread has not completed its transfer and unlocked the buffer, so its processor must have failed, and therefore the buffer is automatically released.

As well as timeouts, an I/O number (an index into a table) is associated with each I/O in order to find its result message, and an I/O state is used to avoid duplication in the case of rollback. Initially, the I/O state is equated to *not-done*. At completion time, it is set to *done*, and the result message is stored in the table. An in-progress state is not necessary, for for several reasons:

(a) Once the non-standard thread runs on the I/O processor it can't be rolled back,

(b) If the non-standard thread is rolled back on the original computing-only processor while its alter-ego on the I/O processor is performing the I/O on the device, forward error recovery actions are used to reflect this, and

(c) Once the non-standard thread has run on the I/O processor, it can recognize and ignore any duplication that might arise from a subsequent rollback on the computing-only processor (see Section 7.9.4.4).

Before looking at the algorithm, take a look at Figure 7.33. It shows the SM and *NRM* data structures involved, and some of the functions used. These functions (represented by arrows) are called by the non-standard thread on a computing-only processor.

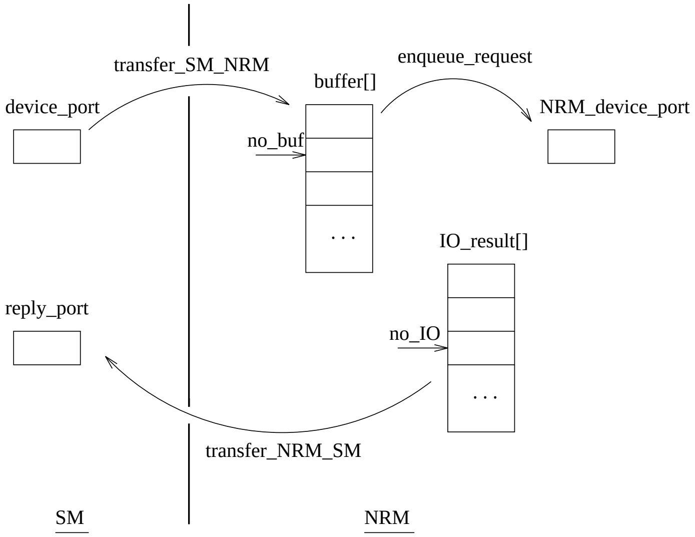


Figure 7.33: SM and *NRM* data structures

```
struct buf{
        lock_t lock;                /*a buffer is used in mutual exclusion */
        int no_io;                  /*number of the associated I/O request */
        char *data;                 /*points to the buffer                 */
} buffer[B];                        /*B buffers are available in the system*/
```

```
transfer_SM_NRM(req,no_buf)         /*returns an I/O number                */
msg_t req;                          /*input: request message               */
int no_buf;                         /*output: allocated buffer number      */
{ char *adr;int no_io;
  no_io=alloc_no_io();              /*allocate an I/O number               */
  no_buf=alloc_buffer(no_io,&adr);  /*allocate a buffer                    */
  copy(req,adr);                    /*data copy                            */
  set_state(no_io,not_done);        /*set the I/O state                    */
  return(no_io);
}

alloc_buffer(no_io,adr)             /*returns the allocated buffer number  */
int no_io;                          /*input: I/O number                    */
char *adr;                          /*output: message buffer address       */
{ for (i=0;i<B;i++%B)               /*loops until a buffer is allocated    */
     if test_lock(buffer[i].lock)
        /*returns false if locked, else returns true and locks it          */
        break;                      /*a free buffer is found               */
  set_timeout(i);                   /*set the timeout for the buffer       */
  adr=buffer[i].data;               /*give the message buffer address      */
  buffer[i].no_io=no_io;            /*set the I/O number                   */
  return(i);
}
```

** Enqueue_request

This routine inserts the request message (which has been stored in a buffer) in the *NRM* device port. Since it operates on the *NRM* state, it must not be disturbed by a commit or rollback. A commit is inconvenient because it may not be possible to say how long it will take, and then the buffer timeout could expire before the message has been inserted into the device port, causing the I/O request to be lost. Similarly with rollback, since the *NRM* state would not be completely updated and then would not be consistent. Therefore, commits and rollbacks have to be disabled while the enqueueing takes place.

```
enqueue_request(no_buf,no_io)
{ if (locked(buffer[no_buf].lock)&&(buffer[no_buf].no_io==no_io))
      { mask_C/R();           /*mask commit and rollback instructions    */
        send(NRM_device_port,buffer[no_buf].data);
                  /*insert the request message in the device port in NRM */
        unlock(buffer[no_buf].lock);            /*free the buffer        */
        buffer[no_buf].no_io=-1;                /*erase the I/O number   */
        unmask_C/R();        /*unmask commit and rollback instructions   */
   }
}
```

** Lock_on_master

To force the non-standard thread to run on the I/O processor, the current processor identity is saved in a new saved_processor field in the data structure associated with threads (in order to more easily find it on the way back), the thread is bound to the I/O processor, and then it is blocked. It will run on the I/O processor after its next scheduling.

```
lock_on_master
{ thread_t thread;
  thread=current_thread();             /*give access to the thread structure */
  thread->saved_processor=thread->bound_processor;
                                       /*save the original processor identity*/
  thread_bind(thread,master_processor);
                                       /*bind the thread to the I/O processor*/
  thread_block((void (*)()) 0);        /*suspend the thread execution        */
}
```

∗∗ Transfer_NRM_SM

On the I/O processor, an I/O operation is performed and its result is stored in a table named `IO_result`, indexed by the I/O number. This message then has to be copied from the *NRM* to the SM. If a processor fails while performing the transfer, then we just have to restart the copy from this table once the non-standard thread has rolled back. The non-standard thread will restart its execution at its last recovery point, i.e. just before the `enqueue_request` function call; it will again be migrated to and locked on the I/O processor, will be ignored (see the description of `enqueue_request`), and then be unlocked and will finally call the `transfer_NRM_SM` routine again.

```
struct store_result{
    int state;                          /*the state is set as not_done or done */
    char *result;                       /*points to the I/O result message     */
} IO_result[N];

transfer_NRM_SM(no)           /*returns a pointer to the I/O result message*/
int no;                             /*no is the I/O number                 */
{ msg_t result;
  copy(IO_result[no].result,&result);  /*data copy                          */
  return(result);
}
```

∗∗ Notify_ack

After the non-standard thread has sent the result message to the reply port associated with the device, and committed, the result can be released by removing the entry from the `IO_result` table.

```
notify_ack(no)
int no;                             /*no is the I/O number                 */
{ char *pt;
   if (pt=IO_result[no].result) != NIL
      { free(pt);                   /*free the result buffer               */
        pt=NIL;
      }
}
```

7.9.4.4 Algorithms for the non-standard thread running on the I/O processor

When the non-standard thread is locked on the I/O processor, it performs the I/O operation on the device and saves the result message in the `IO_result` table. It does not access to the SM, so as not to create any dependency, and therefore rollbacks of the I/O processor will not occur. The I/O processor is dedicated to I/O operations on devices, and we have built a firewall between this I/O processor and the rest of the system.

```
set_state(no,state)                /*set the I/O state in the IO_result table  */
int no;
int state;
{ IO_result[no].state=state;
}

get_state(no)                      /*get the I/O state from the IO_result table*/
int no;
{ return(IO_result[no].state);
}

unlock_master                      /*lock a thread on its original processor   */
{ thread_t thread;
  thread=NRM_current_thread();     /*give access to the thread structure in NRM*/
  thread_bind(thread,thread->saved_processor);
                                   /*bind the thread to its original processor*/
  thread_block((void (*)()) 0);    /*suspend the thread execution              */
}

        ...                         /*code executed by the NST on the     */
        ...                         /*computing_only processor            */
        lock_on_master()           /*run the NST on the I/O processor     */
```

```
if get_state(no_io)==not_done
   { receive(NRM_device_port,request);
     perform the I/O operation;
     IO_result[no_io].result=malloc(sizeof(result));
     save(IO_result[no_io].result, result);
     set_state(no_io,done);
   }
unlock_master();
...                       /*the NST runs again on its original  */
...                       /*processor                           */
```

The thread first checks whether the I/O has already been performed. This is necessary because a processor failure after the non-standard thread has been unlocked from the I/O processor (while the non-standard thread executes the `transfer_NRM_SM` routine or the message sending) will cause the non-standard thread to again be migrated to and locked on the I/O processor, but in this case, the I/O operation has already been performed and its result stored, so it needn't be performed again.

If the I/O state is not equal to *done* (i.e. this is the first time the non-standard thread has executed this I/O operation on the I/O processor), then the non-standard thread has to perform the I/O on the device as described in Section 7.9.3. The result is stored in the `IO_result` table, and the I/O state is then set to *done*. Finally, the non-standard thread is unlocked from the I/O processor.

7.9.4.5 Scheduling and forward error recovery actions

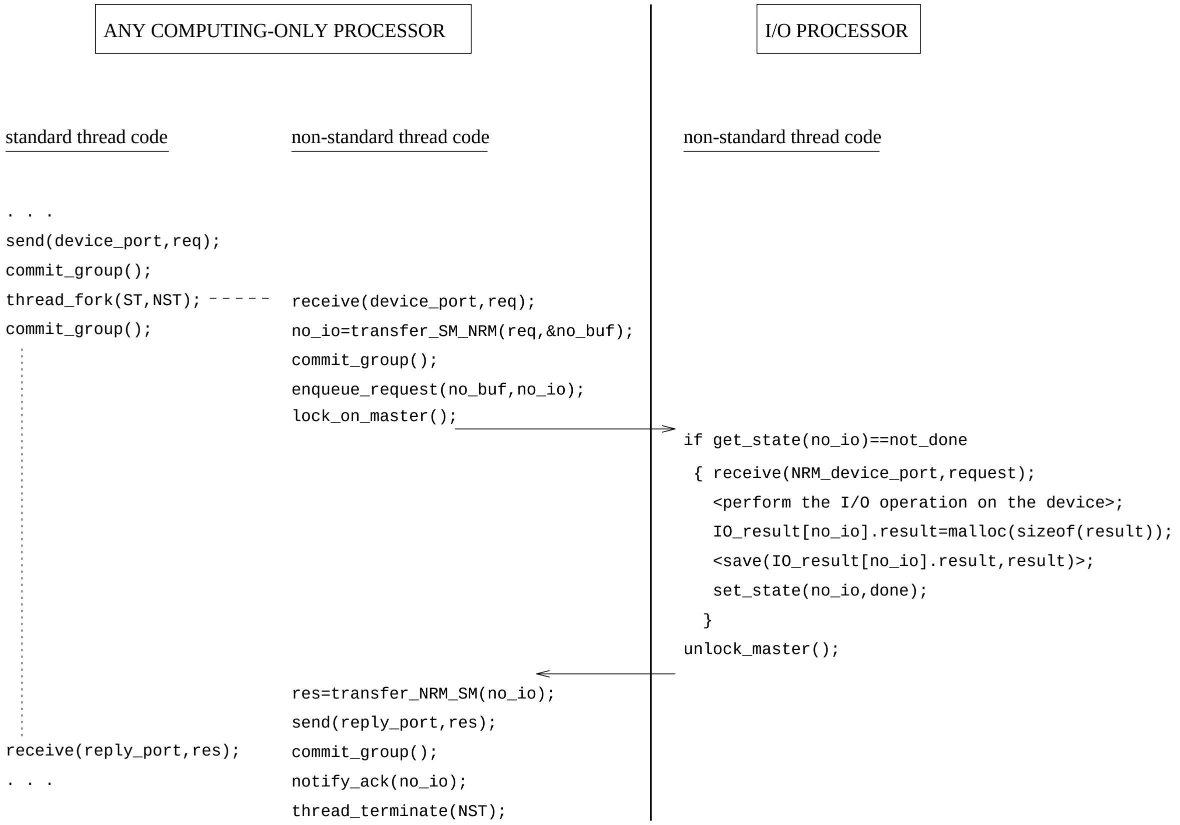
Figure 7.34 illustrates the I/O development on the different processors involved. The computing-only processors have all their data in the SM, including their run queues and threads state. The I/O processor has all its context in *NRM*, in particular its run queue. When the non-standard thread running on the computing-only processor calls the `lock_on_master` routine, the thread is re-scheduled on the I/O processor. To do this, the `lock_on_master` function uses the `current_thread` routine to get a pointer to the thread structure, updates its `saved_processor` and `bound_processor` fields, and then blocks the thread execution. At the next scheduling, the thread structure is removed from the computing-only processor's run queue and inserted into the I/O processor's.

In order that the I/O processor does not become dependent on any computing-only processor, it should not access the SM. Therefore, the thread structures corresponding to the threads running on the I/O processor should be resident in *NRM*. This requires the scheduler to be modified so that it duplicates the thread structure in *NRM* when it moves from the computing-only processor's run queue to the I/O processor's.

On the other side of the coin, when the thread moves at completion time from the I/O processor to the computing-only processor, the scheduler has to update the `thread_t` structure in the SM accordingly. This is why the `unlock` routine only modifies the `thread_t` structure in the *NRM* and not that in the SM. After the `thread_t` structure has been updated in the SM, the one in *NRM* can be released.

If a rollback arises while the non-standard thread is performing the I/O on the device, we are faced with the following situation: on the one hand the non-standard thread keeps on performing the I/O on the I/O processor, and on the other the computing-only processor is rolled back, thereby restoring its run queue to a state in which the non-standard thread is still present. Assuming the I/O is non-idempotent, it is obvious that the thread on the computing-only processor will have to be adjusted somehow, allowing that on the I/O processor to continue undisturbed. One solution is to use some forward error recovery actions here so that the system reflects the fact that the non-standard thread is now running on the I/O processor. A simple way to deduce which `thread_t` structures have to be updated is to examine the I/O processor run queue so as to modify the corresponding structures in the SM accordingly, i.e. to remove the `bound_processor` field and the `thread_t` structure from the computing-only processor run queue. The operation could be simplified if the *NRM* `thread_t` structure included a pointer to the corresponding structure in the SM, as shown in Figure 7.35.

If the non-standard thread has completed the I/O operation on the device and returned to its original processor when the rollback arises, then it is no longer present in the I/O processor run queue. The computing-only processor run queue is restored with the non-standard thread `thread_t` structure chained within. Here, no forward error recovery action is necessary: the non-standard thread restarts its execution on the computing-only processor and is migrated to and locked on the I/O processor after its next call to the `lock_on_master` routine. The I/O operation is not duplicated because the I/O processor first checks if it has already been performed. The only concern is the transfers between SM and *NRM*.



ANY COMPUTING-ONLY PROCESSOR

standard thread code

```
. . .
send(device_port,req);
commit_group();
thread_fork(ST,NST);
commit_group();

receive(reply_port,res);
. . .
```

non-standard thread code

```
receive(device_port,req);
no_io=transfer_SM_NRM(req,&no_buf);
commit_group();
enqueue_request(no_buf,no_io);
lock_on_master();

res=transfer_NRM_SM(no_io);
send(reply_port,res);
commit_group();
notify_ack(no_io);
thread_terminate(NST);
```

I/O PROCESSOR

non-standard thread code

```
if get_state(no_io)==not_done
 { receive(NRM_device_port,request);
   <perform the I/O operation on the device>;
   IO_result[no_io].result=malloc(sizeof(result));
   <save(IO_result[no_io].result,result)>;
   set_state(no_io,done);
  }
unlock_master();
```

Figure 7.34: An I/O operation development for the first implementation proposal

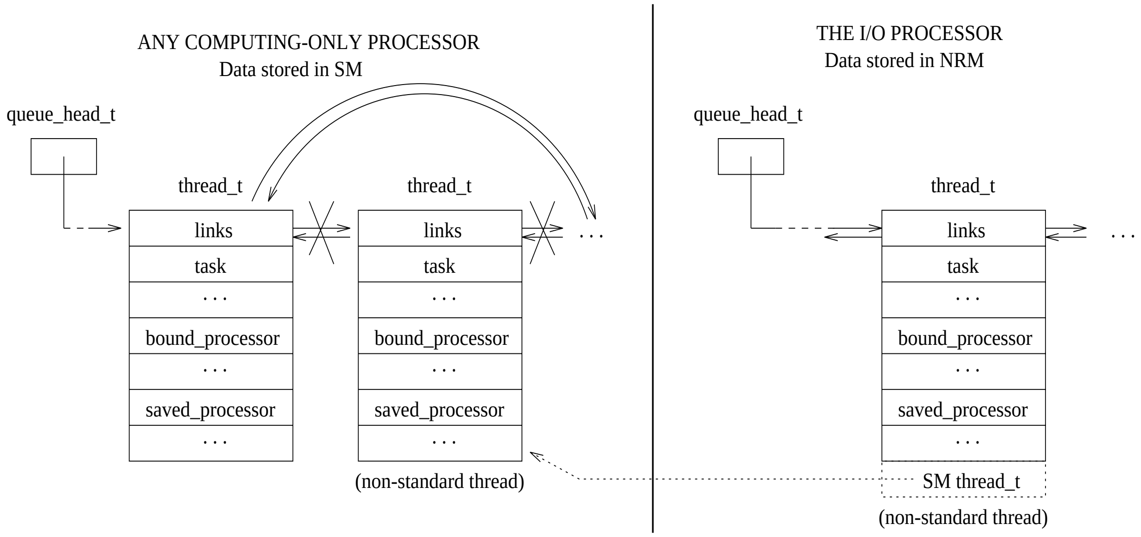


Figure 7.35: Scheduling and forward error recovery actions

In both the above rollback scenarios, the transfers of the `thread_t` structures between SM and *NRM* are performed by the scheduler in the kernel, so they won't be disturbed by commits or rollbacks in the way that user threads would be. The only limitation concerns the data transferred: if a rollback arises while performing the transfer, the data structures should not be recovered until the transfer completes, i.e. any recovery operation should be deferred.

7.9.5 A second implementation proposal for reliable I/O

The first implementation proposal relies on one of the original I/O handling principles of Mach: the thread which requests an I/O operation migrates to the I/O processor to perform it. The transfers between SM and *NRM* (and conversely) are quite crucial. From SM to *NRM*, buffers are necessary to ensure the message insertion in the device port will insert the whole message, or won't insert it at all. For the *NRM* to SM transfer, buffers are not necessary, and a rollback while performing the routine leads the SM to a new state (the one at its last commit point) where the piece of the transferred message has been erased. There is an asymmetry in the behaviour of these two transfers, due to the two different kinds of memory.

Various small things haven't been considered: we haven't precisely specified how to manage timeouts, how to allocate and manage I/O numbers (`alloc_no_io`), or how many buffers should be available (i.e. how large the constant `B` in `transfer_SM_NRM` should be). Those problems need to be studied further for a real implementation. Moreover, we have dedicated the I/O processor exclusively to I/O operations; no standard thread can run on this processor. If we want to allow standard threads to run on it, another run queue should be available, invalidating the scheduling modifications suggested in Section 7.9.4.5. Furthermore, the firewall between I/O operations and the rest of the system would be less strong.

Instead, let us now look at an alternative approach, using a client/server scheme.

7.9.5.1 Principles

Here again, we propose to build a firewall between the computing-only processors and the I/O processor. The I/O processor is dedicated to handling I/O operations, and has a *NRM* associated with it for handling the I/O data structures. The *NRM* is accessible from the computing-only processors, and the I/O processor is able to access the SM. The computing-only processors can be rolled back, but the I/O processor cannot. Therefore, dependencies between recoverable and unrecoverable processors have to be managed so as to support this behaviour.

In this proposal, the I/O processor is a server which is dedicated to I/O operations on devices. This processor has a thread for each device port, each waiting for an I/O request. An I/O operation happens in the following way:

(a) a standard thread, running on any computing-only processor, sends a request to the device port in the SM,

(b) a non-standard thread, running on the I/O processor, receives the request message from the device port, performs the I/O on the device and sends the result message to the reply port,

(c) the standard thread gets the result message from the reply port.

Since the standard thread runs on any computing-only processor, it can support commit and rollback operations. On the other hand, a non-standard thread cannot support any rollback, since it operates on a physical device which is an unrecoverable object. It is, however, able to access the device ports held in the SM and to validate the state of these ports; it does not access any other information in the SM.

Since the non-standard thread is able to access the device ports in the SM, dependencies may be recorded between the I/O processor and the computing-only processors. However, since the I/O processor cannot support rollback operations, we *must* not allow such dependencies to arise.

First, let's examine the port structure, where a dependency may be recorded (see Figure 7.36). This figure illustrates a port containing two messages. The port structure contains two main items of information[13]: a lock and a pointer to a queue of messages inserted into that port. For each message, we have some more information and a pointer to the `io_data` to be transferred, if any (for example, the data to be written for a `device_write`).

It is possible to create a dependency with the port lock held during the message insertion into (and retrieval from) the device port. In order not to create such a dependency, this proposal holds the lock in *NRM*, and the rest of the port structure in the SM. Moreover, this lock allows implementation of the atomic operations presented in Section 7.9.2.2. To use the lock, explicit calls are made to lock and unlock routines, external to the Mach `send` and `receive` primitives, which themselves are modified so as not to manipulate the port locks.

Explicitly locking the port while sending and receiving messages gives an interesting twist to our problem: it prevents other threads from accessing the port before the current one has performed its operation and validated it to erase the dependency created by the access. Since the lock is held in *NRM*, no dependency will be recorded when acquiring or releasing it.

[13] We are not concerned here with the other information.

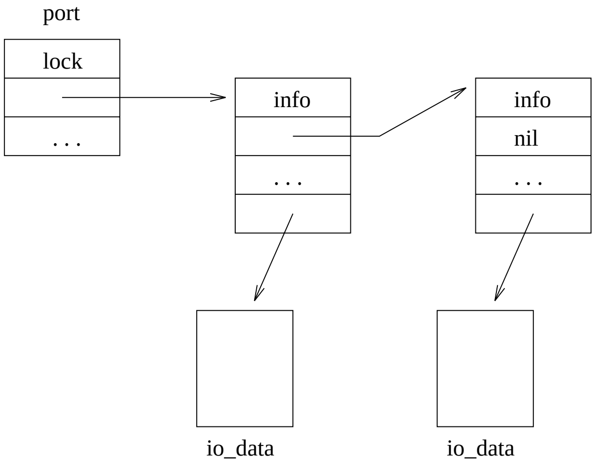
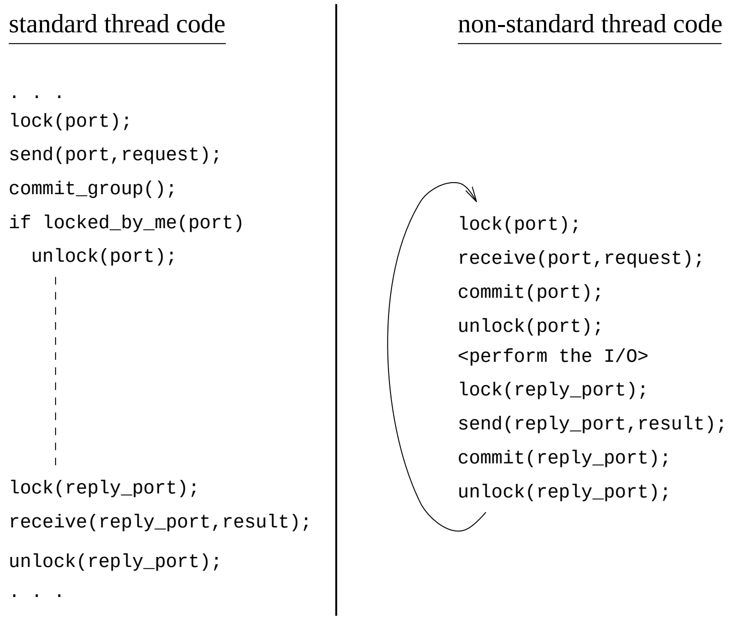


Figure 7.36: A device port structure

7.9.5.2 Algorithmic description

The general I/O development is illustrated in Figure 7.37, again adhering to the general protocol presented in Section 7.9.2.2. The standard thread sends a request message to the device port and commits in an atomic operation (these atomic operations are implemented by means of locks, as explained in the next paragraph). Then the non-standard thread (the server thread associated with this device port) receives this request and validates the new port state in an atomic operation. Next, the non-standard thread performs the physical I/O on the device. At completion time, it sends the result message to the reply port and validates the port state again in an atomic operation. The standard thread can then receive the result message from the reply port.



standard thread code

```
. . .
lock(port);
send(port,request);
commit_group();
if locked_by_me(port)
  unlock(port);

lock(reply_port);
receive(reply_port,result);
unlock(reply_port);
. . .
```

non-standard thread code

```
lock(port);
receive(port,request);
commit(port);
unlock(port);
<perform the I/O>
lock(reply_port);
send(reply_port,result);
commit(reply_port);
unlock(reply_port);
```

Figure 7.37: An I/O operation development for the second implementation proposal

For the non-standard thread running on the I/O processor, the new primitive `commit(zone)`[14] validates the port state in the SM, triggering a copy from the current value to the recovery value. The `commit_group` primitive is not used because the non-standard thread runs on the I/O processor and therefore is not involved in the recovery

[14] introduced in Section 7.9.2.2

protocol or its primitives.

Atomicity is enforced using locks. The indivisibility property is easily enforced, since no other thread can access the port until the corresponding lock has been released by the thread currently holding it. The all or nothing property is enforced through the use of locks and also thanks to the SM design principles, since if a rollback arises while executing the code of the atomic operation, then the last recovery point state is recovered, and either the `commit_group` or `commit(zone)` primitive has been executed, or if not, then all the modifications are undone thanks to the SM rollback. However, if such a rollback does occur, a forward error recovery action is necessary before restarting the thread: the port lock must be released in order to avoid a possible deadlock. This is explained in the next section.

The locking scheme also enforces another important feature for the protocol: it prevents the non-standard thread from receiving a message before it has been validated in the SM. Indeed, the non-standard thread cannot access the device port until the standard thread has unlocked it with `unlock(port)`, since the non-standard thread is blocked waiting for the lock. An important point here is that no other thread can access the port between the `send` or `receive` primitive and the commit or validation.

7.9.5.3 Lock recovery

Remember, in order to avoid creating dependencies with the port lock, it is held in *NRM*. If a rollback arises during execution of an atomic operation (i.e. before the lock has been released), then the standard thread will restart its execution from its last recovery point. But since the lock is held in *NRM*, it will remain locked and will no longer be acquirable. One simple solution consists of releasing the lock before restarting the execution, although this requires the following information to be associated with the lock in *NRM*:

(a) *proc*: processor identifier to detect if the processor supporting the thread that holds the lock (if any) belongs to the dependency group or not (this is used for the forward error recovery action). If the lock isn't currently held the value of *proc* is nil.

(b) *thr*: thread identifier that is checked (when a thread wants to release the lock) to see whether this thread really holds it, since the lock may have been released in the meantime by a forward error recovery action, and then obtained by another thread, in which case it must not be released.

The lock and these two associated items can be maintained within a structure for each device port and reply port, and all these structures can be linked together so as to increase the forward error recovery action performance. After a rollback operation, the forward error recovery action consists of examining the list of port locks, and for each one locked by a processor which belongs to the dependency group, releasing the lock and resetting the two associated items. Then the execution can be restarted.

The second item in (b) above concerns the first atomic operation performed by the standard thread. A commit operation is performed during this atomic operation. This means that subsequent rollbacks (arising before the next commit operation) will cause the standard thread to restart its execution from the instruction following the `commit_group` primitive. This instruction aims to unlock the port, but the lock may by then be held by another thread, therefore the current thread first checks if it is holding the lock before calling the `unlock` primitive, instead of releasing it automatically. Thus the first standard thread atomic operation is encoded in the following way:

```
lock(port);
send(port,request);
commit_group();
if locked_by_me(port)
   unlock(port);
```

The code of the `locked_by_me` primitive is as follows:

```
locked_by_me(port)
port_t port;                                 /*port identifier*/
{ return (locked(port) && (port->proc==current_processor())
                        && (port->thr==current_thread()));
}
```

The second atomic operation performed by the standard thread is simpler since it does not involve a commit. Similarly, the non-standard thread is not involved in the recovery protocol, so it does not face such problems.

7.9.6 Comments

The second implementation proposal relies on a client/server scheme for I/O handling. The standard threads run on the computing-only processors while the non-standard threads run on the I/O processor (the server). Dependencies between these two types of threads are handled according to the general I/O protocol principles. The algorithms come directly from the coding of this general protocol. The main feature of this coding consists of holding the port locks in *NRM* so as to support an implementation of the atomic operations. Here again, we have dedicated the I/O processor exclusively to I/O operations, thereby preventing any standard thread from running on this processor.

Thus both implementation proposals adopt the proposed general protocol for communicating with devices, at the expense of adding a non-recoverable memory to the architecture to support the I/O processor data management. While the first implementation proposal is based on transfers between the two types of memories, the second one relies on an implementation of the atomic operations; both aim to handle dependencies between recoverable and unrecoverable objects. Both approaches introduce extra commits, but this is not expected to cause performance problems because few things are expected to be modified between checkpoints, so commits won't take too much time. Moreover, for the non-standard thread, these commit operations aim to validate the port state and may be simplified.

Two important points are satisfied with the I/O protocol: both loss and duplication of I/O requests are avoided in a simple manner. Therefore a new layer is not needed to resend lost messages or to discard duplicated messages. Moreover, the modifications proposed apply to different devices independently of their type. The major drawback is that we have assumed that the I/O processor never fails, nor do any devices, which poses yet another unresolved issue.

Bibliography

[Accetta 86] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer USENIX Conference*, pp.93–112, Atlanta, Georgia, June 1986.

[Amdahl 67] AMDAHL, G.M. Validity of the Single Processor Approach to achieving Large Scale Computing Capabilities. *Proceedings of AFIPS 1967 Spring Joint Computer Conference*, pp.483–485, Atlantic City, New Jersey (NJ), April 1967.

[Ahmed et al 90] AHMED, R., FRAZIER, R., AND MARINOS, P. Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems. *Proceedings of 20th International Symposium on Fault-Tolerant Computing Systems*, pp.82–88, Newcastle, June 1990.

[Anderson et al 81] ANDERSON, T., AND LEE, P.A. *Fault-Tolerance : Principles and Practise*, Prentice Hall, 1981.

[Archibald 86] ARCHIBALD, J. High Performance Cache Coherence Protocols for Shared-Bus Multiprocessors. *Technical Report 86-06-02*, Computer Science Department, University of Washington, 1986.

[Archibald et al 86] ARCHIBALD, J., AND BAER, J.L. Cache Coherence Protocols : Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, pp.273–298, Vol.4, No.4, November 1986.

[Babaoglu 90] BABAOGLU, O. Fault-Tolerant Computing Based on Mach. *Operating System Review*, pp.27–39, Vol.24, No.1, January 1990.

[Bach 86] BACH, M. *The Design of the UNIX Operating System*, First Edition, Prentice Hall, 1986.

[Banâtre et al 86] BANÂTRE, J.P., BANÂTRE, M., LAPALME, G., AND PLOYETTE, F. The Design and Building of Enchere, a Distributed Electronic Marketing System. *Communications of the ACM*, pp.19–29, Vol.29, No.1, January 1986.

[Banâtre et al 88a] BANÂTRE, J.P., BANÂTRE, M., AND MULLER, G. Ensuring Data Security and Integrity with a Fast Stable Storage. *Proceedings of 4th International Conference on Data Engineering*, pp.285–293, Los Angeles, February 1988.

[Banâtre et al 88b] BANÂTRE, M., BANÂTRE, J.P., PLOYETTE, F., DECOUTY, B., AND PRUNAULT, Y. Apparatus and Method for Fast and Stable Data Storage. *United States Patent 4,734,855*, INRIA, March 1988.

[Banâtre et al 90a] BANÂTRE, M., AND JOUBERT, P. Cache Management in a Tightly Coupled Fault Tolerant Multiprocessor. *Proceedings of 20th International Symposium on Fault-Tolerant Computing Systems*, pp.89–96, Newcastle, June 1990.

[Banâtre et al 90b] BANÂTRE, M., AND JOUBERT, P. A Fault Tolerant Tightly Coupled Multiprocessor Architecture based on Stable Transactional Memory. *Technical Report 1178*, INRIA, March 1990.

[Banâtre et al 90c] BANÂTRE, M., MORIN, C., MULLER, G., ROCHAT, B., AND SANCHEZ, P. Stable Transactional Memories and Fault Tolerant Architectures. *Proceedings of 4th ACM SIGOPS European Workshop, Fault Tolerance Support in Distributed Systems*, Bologne, Italy, September 1990.

[Banâtre et al 91a] Banâtre, M., Morin, C., Muller, G., Rochat, B., and Sanchez, P. Stable Transactional Memories and Fault Tolerant Architectures. *OS Review*, January 1991.

[Banâtre et al 91b] Banâtre, M., Muller, G., Rochat, B., and Sanchez, P. Design Decisions for the FTM: A General Purpose Fault Tolerant Machine. *Proceedings of 21st International Symposium on Fault-Tolerant Computing Systems*, pp.71–78, Montréal, Canada, June 1991.

[Banâtre et al 91c] Banâtre, M., Heng, P., Muller, G., and Rochat, B. How to Design Reliable Servers using Fault Tolerant Micro-Kernel Mechanisms. *USENIX Mach Symposium*, pp.223–231, Monterey, California, November 1991.

[Banâtre et al 92a] Banâtre, M., Jégado, M., Joubert, P., and Morin, C. Communicating Processes and Fault Tolerance: A Shared Memory Multiprocessor Experience. *Research Report No.1649*, INRIA, March 1992.

[Banâtre et al 92b] Banâtre, M., Jégado, M., and Joubert, P. Communicating Processes and Fault Tolerance: A Shared Memory Multiprocessor Experience. *Technical Report No.647*, INRIA, March 1992.

[Barbou des Places et al 94] Barbou des Places, F., Bernardat, P., Condict, M., Empereur, S., Febvre, J., George, D., Loveluck, J., McManus, E., Patience, S., Rogado, J., and Rodaud, P. Architecture and Benefits of a Multithreaded OSF/1 Server. *Technical Report*, Open Software Foundation, 1994.

[Baron et al 88] Baron, R., Black, D., Bolosky, W., Chew, J., Golub, D., Rashid, R., Tevanian, A., and Young, M. *MACH Kernel Interface Manual*, Department of Computer Science, Carnegie Mellon University, September 1988.

[Bartlett et al 87] Bartlett, J., Gray, J., and Horst, B. Fault Tolerance in Tandem Computer Systems. In *The Evolution of Fault-Tolerant Computing*, Edited by Avizienis, A., Kopetz, H., and Laprie, J.C., pp 55–76, Vol.1, Springer Verlag, 1987.

[Bartlett et al 90] Bartlett, J., Carr, R., and Garca, D. Fault Tolerance in Tandem Computer Systems. *Technical Report 90.5*, Tandem, March 1990.

[Bell 92] Bell, D., and Grimson, J.B. *Distributed Database Systems*, Addison-Wesley, 1992.

[Bernstein et al 87] Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bernstein 88] Bernstein, P.A. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, pp.37–45, Vol.21, February 1988.

[Best 82] Best, E. Relational Semantics of Concurrent Programs (with some Applications). *Technical Report No.180*, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, U.K., July 1982.

[Bjørner et al 78] Bjørner, D., et al., Editors The Vienna Development Method: The Meta-Language. *Lecture Notes in Computer Science*, Vol.61, Springer-Verlag, 1978.

[Bjørner et al 91] Bjørner, D., and Jones, C.B., Editors VDM'87: VDM- A Formal Method at work. *Lecture Notes in Computer Science*, Vol.252, Springer-Verlag, 1991.

[Black 90] Black, D. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE*, May 1990.

[Borg et al 89] Borg, A., Blau, W., Graetsch, W., Herrmann, F., and Oberle, W. Fault Tolerance under UNIX. *ACM Transactions on Computer Systems*, pp.1–24, Vol.7, No.1, 1989.

[Bowen et al 93] Bowen, N.S., and Pradhan, D.K. Processor- and Memory-Based Checkpoint and Rollback Recovery. *IEEE Computer*, pp.22-31, February 1993.

[Butterfield 93] BUTTERFIELD, A. A VDM study of Fault-Tolerant StableStorage - towards a computer engineering mathematics. In *FME'93: Industrial-Strength Formal Methods*, Vol.670 of *Lecture Notes in Computer Science*, pp.216-234, Springer-Verlag, April 1993.

[Carter et al 71] CARTER, W., AND BOURICIUS, W. A Survey of Fault Tolerant Computer Architecture and its Evaluation. *Computer*, pp.9–16, Vol.4, No.1, January 1971.

[Censier et al 78] CENSIER, L.M., AND FEAUTRIER, P. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, pp.1112–1118, Vol.27, No.12, December 1978.

[Chandy et al 93] CHANDY, J., AND REDDY, A.L.N. Failure Evaluation of Disk Array Organization. *Proceedings of the International Conference on Distributed Computing Systems*, pp.319–326, IEEE Computer Society, 1993.

[Chen et al 95] CHEN, P.M., AND LEE, E.K. Striping in a RAID Level 5 Disk Array. *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp.136—145, May 1995.

[Chervenak 90] CHERVENAK, A.L. Performance Measurements of the First RAID Prototype. *Technical Report UCB/CSD 90/674*, University of California, May 1990.

[Chervenak et al 91] CHERVENAK, A.L., AND KATZ, R.H. Performance of a RAID Prototype. *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, San Diego, California (CA), May 1991.

[Chervenak 91] CHERVENAK, A.L. Performance of a Disk Array Prototype. *CMG Transactions*, 1991.

[Clatin et al 93] CLATIN, M., MORIN, C., MULLER, G., AND ROUTEAU, J.P. Implementing an Emulation of FASST Stable Memory using the Cthread Package. *Technical Report*, INRIA, October 1993.

[Coghlan et al 89] COGHLAN, B.A., AND JONES, J.O. Improvements in and relating to stable memory circuits. *Irish Patent Application 2223/89 and derivatives*, Trinity College Dublin and Tolsys Ltd., September 1989.

[Coghlan et al 91] COGHLAN, B.A., AND JONES, J.O. Stable Memory Operations. *Irish Patent Application 1094/91 and derivatives*, Tolsys Ltd., April 1991.

[Coghlan et al 92a] COGHLAN, B.A., JONES, J.O., OCARROLL, P., MCGRATH, P., AND HENNESSY, B. The Stable Disk. *Technical Report*, Esprit Project P5212 (FASST), September 1992.

[Coghlan et al 92b] COGHLAN, B.A., AND JONES, J.O. Memory Checkpointing. *Irish Patent Application 2784/92 and derivatives*, Tolsys Ltd., November 1992.

[Coghlan et al 93] COGHLAN, B.A., JONES, J.O., CHUNG, H.V., LABOD, T., AND BRYCE, C.J. Advanced Multiprocessor Architectures. *Final Report for EOLAS Grant No.ST/304/89*, Department of Computer Science, Trinity College Dublin, December 1993.

[Coghlan et al 94] COGHLAN, B.A., AND JONES, J.O. Stable Memory for a Disk Write Cache. *Microprocessing and Microprogramming*, pp.53–80, Vol.41, 1995.

[Copeland 89] COPELAND, G., KELLER, T., KRISNAMURTHY, R., AND SMITH, M. The Case for Safe RAM. *Proceedings of 15th Very Large DataBase Conference*, 1989.

[Corollary 92] Corollary 486/smp and 486/smpXM, System Installation and Maintenance. *Technical Report*, Corollary Inc., 1992.

[Cray 85] Cray X-MP and Cray-1 Computer Systems: Disk Systems Hardware Reference Manual, Vol.H0077. *Cray Research*, 1440 Northland Drive, Mendota Heights, Minnesota (MN), 1985.

[CSR 91a] IEEE P1212 WORKING GROUP CSR Architecture (Control and Status Register Architecture). *IEEE*, August 1991.

[CSR 91b] IEEE P1212.1 Working Group CSR Architecture (DMA Framework). *IEEE*, November 1991.

[Davis et al 91] Davis, H., Goldschmidt, S., and Hennessy, J. Multiprocessor Simulation using Tango. *Proceedings of 1991 International Conference on Parallel Processing*, pp.II 99–107, August 1991.

[DT-Connect] Data Translation *DT-Connect Specification*, Data Translation Ltd..

[DT-Connect II] Data Translation *DT-Connect-II Specification*, Data Translation Ltd..

[EISA 92] EISA *Extended Industry Standard Architecture (EISA) Specification*, Version 3.12, BCPR Services Inc., New York, 1992.

[Eppinger 1] Eppinger, J.L., Mummert, L.B., and Spector, A.Z., Editors *Camelot and Avalon - A Distributed Transaction Facility*, ISBN 1-55860-185-6, Morgan Kaufman.

[Fabregat et al 94] Fabregat, G., Pérez, C., Boluda, J.A., Martínez, R.J., and Munt, R. The FASST DPU Technical Description. *Technical Report*, Esprit Project P5212 (FASST), July 1994.

[FASST 89] FASST Consortium FASST - Fault tolerant Architecture with Stable Storage Technology. *Technical Annex, FASST Document No.6*, Esprit Project P5212 (FASST), December 1989.

[FASST 92] FASST Consortium The FASST Architecture: Overall Requirements and Specifications. *Technical Report*, Esprit Project P5212 (FASST), January 1992.

[Ferrara 91] Ferrara, J., Editor *CALYX Systems Specification*, Digital Equipment Corporation, May 1991.

[Fielland et al 84] Fielland, G., and Rodgers, D. 32-Bit Computer System Shares Load Equally Among up to 12 Processors. *Electronic Design*, pp.153–168, January 1984.

[Finley 90] Finley, R.F. *European Patent Specification No. EP-Bl-0, 119, 806*, 1990.

[Frank 84] Frank, S.C. Tightly Coupled Multiprocessor Systems Speed Memory Access Times. *Electronic Design*, pp.164–169, Vol.57, January 1984.

[Futurebus+ 90] IEEE P896.1 Working Group Futurebus+ P896.1, Logical Layer and Specifications. *IEEE Computer Society Press*, February 1990.

[Futurebus+ 91] IEEE P896.2 Working Group Futurebus+ P896.2, Physical Layer and Profile Specification. *IEEE Computer Society Press*, January 1991.

[Futurebus+ 94a] ISO/IEC 10857:1994 (ANSI/IEEE Std. 896.1) Futurebus+ Standard, 1994 Edition. *IEEE*, 1994.

[Futurebus+ 94b] IEEE High-Performance I/O Bus Architecture: a Handbook for IEEE Futurebus+ Profile B. *IEEE Standards Press*, 1994.

[Gabbe et al 84] Gabbe, J.D., and Hecht, M.S. *United States Patent Specification No. US-A-4, 459, 658*, 1984.

[Gefflaut 92] Gefflaut, A., and Joubert, P. SPAM : A Multiprocessor Execution Driven Simulation Kernel. *Research Report*, INRIA, December 1992.

[Gien 90] Gien, M. Micro-Kernel Architecture: Key to Modern Operating System Design. *UNIX Review*, pp.10, November 1990.

[Gil et al 97] Gil, P., Baraza, J., Gil, D., and Serrano, J. High Speed Fault Injector for Safety Validation of Industrial Machinery. *Proceedings of the 8th European Workshop on Dependable Computing*, 1997.

[Goodman 84] Goodman, J.R. Using Cache Memory to Reduce Processor-Memory Traffic. *Proceedings 10th International Symposium on Computer Architecture*, pp.124–131, Stockholm, 1984.

[Goodman 87] GOODMAN, J.R. Cache Memory Optimization to Reduce Processor / Memory Traffic. *Journal of VLSI and Computer Systems*, pp.61–86, Vol.2, No.1, 1987.

[Gray 78] GRAY, J. *Notes on Database Operating Systems.*, Vol.60 of *Lecture Notes in Computer Science*, pp.394–481, Springer Verlag, 1978.

[Gray et al 93] GRAY, J., AND REUTER, A. *Transaction Processing : Concepts and Techniques*, ISBN 1-55860-190-2, Morgan Kauffmann Publishers, San Mateo, California, 1993.

[Greenberg et al 88] GREENBERG, A.G., MITRANI, I., AND RUDOLPH, L. Analysis of Snooping Caches. *Performance'87*, Edited by Courtois, P.-J., and Latouche, G., North-Holland, 1988.

[Gries 81] GRIES, D. *The Science of Programming*, Springer Verlag, New York, 1981.

[Gunneflo 89] GUNNEFLO, U., KARLSSON, J., AND TORIN, J. Evaluation of Error Detection Schemes using Fault Injection by Heavy-Ion Radiation. *Proceedings of 19th International Symposium on Fault-Tolerant Computing Systems*, pp.340–347, Chicago, June 1989.

[Hamming 50] HAMMING, W.R. Error Detecting and Error Correcting Codes. *Bell Systems Technical Journal*, pp.147–160, Vol.29, No.2, April 1950.

[Handy 93] HANDY, J. The Cache Memory Book. *Academic Press*, 1993.

[Harrison et al 87] HARRISON, E.S., AND SCHMITT, E. The Structure of SYSTEM/88, a Fault-Tolerant Computer. *IBM Systems Journal*, pp.293–318, Vol.26, No.3, 1987.

[Heinrich 94] HEINRICH, J. *MIPS R4000 Microprocessor Users Manual*, Second Edition, MIPS Technologies Inc., 1994.

[Henn 89] HENN, R. Feasible Processor Allocation in a Hard Real Time Environment. *The Real-Time Systems Journal*, pp.77–93, Vol.1, No.1, June 1989.

[Holland et al 92] HOLLAND, M., AND GIBSON, G. Parity Declustering for Continuous Operation in Redundant Disk Arrays. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pp.23–35, 1992.

[Howles 94] HOWLES, F. Distributed Arbitration in the IEEE Futurebus Protocol. *Masters Thesis*, Oxford University Computing Laboratory, 1994.

[Horning et al 74] HORNING, J.J., LAUER, H.C., MELLIAR-SMITH, P.M., AND RANDELL, B. A Program Structure for Error Detection and Recovery. *International Symposium on Operating Systems*, pp.171–187, Rocquencourt, France, April 1974.

[Hsueh et al 97] HSUEH, M., TSAI, T., AND IYER, R. Fault Injection: Techniques and Tools. *IEEE Computer*, April 1997.

[Iyer 95] IYER, R. Experimental Evaluation. *Fault Tolerant Computing, FTCS-25 Silver Jubilee Special Issue*, June 1995.

[InMOS 88] Transputer Reference Manual. *InMOS Limited*, Published by Prentice Hall, 1988.

[InMOS 90] Transputer Development System. *InMOS Limited*, Second Edition, Published by Prentice Hall, 1988.

[Janssens et al 91] JANSSENS, B., AND FUCHS, W. Experimental Evaluation of Multiprocessor Cache-based Error Recovery. *Proceedings of 1991 International Conference on Parallel Processing*, pp.I 505–508, August 1991.

[Jewett 91] JEWETT, D. Integrity s2: A Fault-Tolerant UNIX Platform. *Proceedings of 21st International Symposium on Fault-Tolerant Computing Systems*, pp.512–519, Montréal, Canada, June 1991.

[Jones 90] JONES, C.B. *Systematic Software Development using VDM*, 2nd Edition, Prentice Hall, 1990.

[Jöhnk et al 92] JÖHNK, M.J., MORIN, C., MCGRATH, P., SCHWARTZ, W., AND THOMAS, A. FASST Microkernel Specification. *Technical Report*, Esprit Project P5212 (FASST), September 1992.

[Johnson 84] JOHNSON, B.W. Fault-tolerant Microprocessor-based Systems. *IEEE Micro*, December 1984.

[Joubert 89] JOUBERT, P. Etude d'une architecture multiprocesseur tolérante aux fautes. *Technical Report*, INRIA, June 1989.

[Joubert 91] JOUBERT, P. A Fault Tolerant Shared Memory Multiprocessor based on Stable Transactional Memory. *Proceedings of 2nd Workshop on Scalable Shared-Memory Multiprocessors*, May 1991.

[Joubert 93] JOUBERT, P. Conception et évaluation d'une architecture multiprocesseur à mémoire partagée tolérante aux fautes. *Ph.D. thesis*, University of Rennes I, January 1993.

[Katz et al 85] KATZ, R.H., EGGERS, S.J., WOOD, D.A., PERKINS, C.L., AND SHELDON, R.G. Implementing a Cache Consistency Protocol. *Proceedings of 12th Annual International Symposium on Computer Architecture*, pp.276–283, IEEE, Boston, 1985.

[Katz et al 89] KATZ, R.H., GIBSON, G.A., AND PATTERSON, D.A. Disk System Architectures for High Performance Computing. *Proceedings of IEEE*, pp.1842–1858, Vol.77, No.12, December 1989.

[Katzman 78] KATZMAN, J. A Fault-Tolerant Computing System. *Proceedings of 11th Hawaii International Conference on System Sciences*, pp.85–102, Honolulu (HA), January 1978.

[Kim 85] KIM, M.Y. Parallel Operation of Magnetic Disk Storage Devices: Synchronized Disk Interleaving. *Proceedings of 4th International Workshop on Data-Base Machines*, 1985.

[Kim et al 85] KIM, M.Y., AND PATEL, A. Error-Correcting Codes for Interleaved Disks with Minimum Redundancy. *Research Report RC11185*, International Business Machines, May 1985.

[Kim 86] KIM, M.Y. Synchronized Disk Interleaving. *IEEE Transactions on Computers*, Vol.C-35, No.11, November 1986.

[Kim et al 87] KIM, M.Y., AND TANTAWI, A.N. Asynchronous Disk Interleaving. *Research Report RC12497*, International Business Machines, January 1987.

[Kim et al 91] KIM, M.Y., AND TANTAWI, A.N. Asynchronous Disk Interleaving: Approximating Access Delays. *IEEE Transactions on Computers*, July 1991.

[Kitagawa 91] KITAGAWA, M. Understanding MBus. In *The Sparc Technical Papers*, Edited by Catanzaro, B., pp.425–442, Springer Verlag, 1991.

[Krakowiak 85] KRAKOWIAK, S. *Principes des systèmes d'exploitation des ordinateurs*, Dunod Informatique, 1985.

[Lampson 81a] LAMPSON, B. Atomic Transactions. In *Distributed Systems and Architecture and Implementation : an Advanced Course*, Vol.105 of *Lecture Notes in Computer Science*, pp.246–265, Springer Verlag, 1981.

[Lampson 81b] LAMPSON, B., EDITOR *Distributed Systems Architecture and Implementation : an Advanced Course*, Vol.105 of *Lecture Notes in Computer Science*, Springer Verlag, 1981.

[Laprie et al 90] LAPRIE, J.C., ARLAT, J., B'EOUNES, C., AND KANOUN, K. Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. *IEEE Computer*, July 1990.

[Laprie 90] LAPRIE, J. Dependability: Basic Concepts and Associated Terminology. *Internal Report No.90055*, Laboratoire d'Automatique et d'Analyse des Systemes, Toulouse, France, March 1990.

[Larus 90] LARUS, J. Abstract execution : A Technique for Efficiently Tracing Programs. *Software Practice and Experience*, Vol.20, No.12, December 1990.

[Lee et al 80] Lee, P.A., Ghani, N., and Heron, K. A Recovery Cache for the PDP-11. *IEEE Transactions on Computers*, pp.546–549, Vol.C-29, No.6, June 1980.

[Lee et al 90] Lee, P.A., and Anderson, T. *Fault Tolerance : Principles and Practice*, Second Revised Edition, Vol.3 of *Dependable Computing and Fault-Tolerant Systems*, Springer Verlag, New York, 1990.

[E.K.Lee 90] Lee, E.K. Software and Performance Issues in the Implementation of a RAID Prototype. *Technical Report UCB/CSD 90/573*, University of California, May 1990.

[E.K.Lee 91] Lee, E.K. Hardware Overview of RAID-II. *UC Berkeley RAID Retreat, Lake Tahoe*, January 1991.

[E.K.Lee et al 91] Lee, E.K., and Katz, R.H. Performance Consequences of Parity Placement in Disk Arrays. *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, Palo Alto, California (CA), April 1991.

[Leffler et al 89] Leffler, S., Mc Kusick, M.K., Karels, M.J., and Quaterman, J.S. *The Design and Implementation of the 4.3 BSD UNIX Operating System*, First Edition, Addison-Wesley, 1989.

[Liu et al 73] Liu, C.L., and J.W. Layland, J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, pp.46–61, Vol.20, No.1, 1973.

[Livercy 78] Livercy, C. *Théorie des Programmes*. Dunod Informatique, 1978.

[Loepere 92a] Loepere, K. Mach 3 Kernel Interfaces. *Technical Report*, Open Software Foundation and Carnegie Mellon University, July 1992.

[Loepere 92b] Loepere, K. Mach 3 Kernel Principles. *Technical Report*, Open Software Foundation and Carnegie Mellon University, July 1992.

[Mac an Airchinnigh 90] Mac an Airchinnigh, M. Conceptual Models and Computing. *Ph.D. thesis*, Trinity College Dublin, Ireland, 1990.

[Mac an Airchinnigh 91] Mac an Airchinnigh, M. The Irish School of VDM. *VDM'91*, Vol.552 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991.

[Madeira et al 90] Madeira, H., Quadros, G., and Silva, J.G. Experimental Evaluation of a set of Simple Error Detection Mechanisms. *Microprocessing and Microprogramming*, pp.513-520, Vol.30, August 1990.

[Mahmood 88] Mahmood, A. Concurrent Error Detection using Watchdog Processors - a Survey. *Transactions on Computers*, pp.160-174, Vol.37, No.2, IEEE, February 1988.

[Martínez 97] Martínez, R.J. Validación Experimental por Injección física de Fallos de la Garantía de Funcionamiento de un Sistema Multiprocesador Tolerante a Fallos. *Ph.D. thesis*, Universitat de València, España, 1997.

[Martínez et al 95] Martínez, R.J., Pérez, C., Fabregat, G, Boluda, J.A., and Pardo, F. DPU: A FB+ Based Fault Tolerant System. *Proceedings of the Open Bus System Symposium '95*, pp.229–236, June 1995.

[Maunder et al 90] Maunder, C.M., and Tulloss, R.E. The Test Access Port and Boundary Scan Architecture. *IEEE Computer Society Press*, 1990.

[McCreight 84] McCreight, E. The Dragon Computer System: An Early Overview. *Technical Report*, Xerox Corporation, September 1984.

[MacDougall 89] MacDougall, M.H. *Simulating Computer Systems*, Edi Press, 1989.

[McKusick 84] McKusick, M.K. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, pp.181-197, Vol.2, No.3, August 1984.

[Merchant et al 92] Merchant, A., and Yu, P.S. Design and Modeling of Clustered RAID. *Research Report RC17572*, International Business Machines, January 1992.

[Mirapuri et al 92] MIRAPURI, S., WOODACRE, M., AND VASSEGHI, N. The Mips R4000 Processor. *IEEE Micro*, pp.10–22, Vol.12, No.2, 1992.

[Mitrani 87] MITRANI, I. *Modelling of Computer and Communication Systems*, Cambridge University Press, 1987.

[Mitrani 95] MITRANI, I. The Spectral Expansion Solution Method for Markov Processes on Lattice Strips. Chapter 13 in *Advances in Queueing*, Edited by Dshalalow, J.H., CRC Press, 1995.

[Mitrani et al 95] MITRANI, I., AND CHAKKA, R. Spectral Expansion Solution for a Class of Markov Models: Application and Comparison with the Matrix-Geometric Method. *Performance Evaluation*, pp.241–260, Vol.23, 1995.

[Morin et al 92] MORIN, C., AND FRANKENFELD, C. FASST Recovery Protocol and Stable Memory Specification. *Technical Report*, Esprit Project P5212 (FASST), September 1992.

[Muntz et al 90] MUNTZ, R.R., AND LUI, J.C.S. Performance Analysis of Disk Arrays under Failure. *Proceedings of 16th International Conference on Very Large Data Bases (VLDB)*, pp.162–173, August 1990.

[National 85] GENIX 4.2 Programmer's Manual. *National Semiconductor Corporation*, Santa Clara, California, April 1985.

[NCR53C916a] NCR 53C916 FSCSI2 Chip Chip Data Manual. *NCR Corporation*, Wichita, Arizona, March 1991.

[NCR53C916b] NCR 53C916 FSCSI2 Chip Programmer's Guide. *NCR Corporation*, Wichita, Arizona, March 1991.

[NCR53C920a] NCR SCSI Data Path Chip Chip Data Manual. *NCR Corporation*, Wichita, Arizona, March 1991.

[NCR53C920b] NCR SCSI Data Path Chip Programmer's Guide. *NCR Corporation*, Wichita, Arizona, March 1991.

[NCR53C932] NCR 53C932 SCSI Bus Extender Chip Data Manual. *NCR Corporation*, Wichita, Arizona, April 1991.

[Ng et al 92] NG, S., AND MATTSON, R. Maintaining Good Performance in Disk Arrays during Failure via Uniform Parity Group Distribution. *Proceedings of the 1st International Conference on High Performance Distributed Computing*, pp.260–269, Syracuse, New York (NY), September 1992.

[Ors 93] ORS CAROT, R. Sistema Multiprocesador Tolerante a Fallos basado en Puntos de Recuperación. *PhD thesis*, Universidad Politécnica de Valencia, November 1993.

[Ors et al 94a] ORS, R., SERRANO, J.J., SANTONJA, V., GIL, P., PÉREZ, A. AND RODRÍGUEZ, S. A Rollback Control Stable Memory Proposal: System Description. *Technical Report*, Esprit Project P5212 (FASST), July 1994.

[Ors et al 94b] ORS, R., SERRANO, J.J., SANTONJA, V., GIL, P., PÉREZ, A. AND RODRÍGUEZ, S. A Rollback Control Stable Memory Proposal: Performance and Dependability Evaluation. *Technical Report*, Esprit Project P5212 (FASST), 1994.

[OSF 92] OPEN SOFTWARE FOUNDATION The Design of the OSF/1 Operating System. *Technical report*, Open Software Foundation, 1992.

[Ousterhout et al 85] OUSTERHOUT, J., ET AL A Trace-Driven Analysis of the Unix 4.2 BSD File System. *Proceedings of ACM Symposium on Operating Principles*, 15-24 December 1985.

[Ousterhout et al 88] OUSTERHOUT, J.K., AND DOUGLIS, F. Beating the I/O Bottleneck: A Case for Log-Structured FileSystems. *Report No.UCB/CSD 88/467*, University of California, 1988.

[Ousterhout et al 89] Ousterhout, J.K., and Rosenblum, M. The Design and Implementation of a Log-Structured File System. *Proceedings of 13th ACM Symposium on Operating Systems Principles*, 1989.

[Patel et al 84] Patel, J.H., and Papamarcos, M. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. *Proceedings of 11th Symposium on Computer Architecture*, pp.348-354, 1984.

[Patterson et al 87] Patterson, D.A., Gibson G.A., and Katz, R.H. A Case for Redundant Arrays of Inexpensive disks (RAID). *Technical Report UCB/CSD 87/39*, University of California, June 1988.

[Patterson et al 88] Patterson, D.A., Gibson G.A., and Katz, R.H. A Case for Redundant Arrays of Inexpensive disks (RAID). *Proceedings of ACM SIGMOD 88*, pp.109-116, Chicago, Illinois (IL), June 1988.

[Patterson et al 89] Patterson, D.A., Gibson G., and Katz, R.H. Introduction to Redundant Arrays of Inexpensive disks (RAID). *Proceedings of IEEE COMPCON*, Spring 1989.

[Pérez et al 91] Pérez, A., and Rodríguez, S. Real Time Support Executive Facilities. *Technical report*, Universidad Politécnica de Madrid, 1991.

[Pérez et al 94a] Pérez, A., and Rodríguez, S. RSX Design and Specification on OSF/1 MK 4.0.1. *Technical Report*, Universidad Politécnica de Madrid, 1994.

[Pérez et al 94b] Pérez, A., Rodríguez, S., Muñoz, L.M., García, A., Liébana, M.A., and Prieto, L. Real Time Support Executive Implementation. *Technical Report*, Universidad Politécnica de Madrid, 1994.

[POSIX 88] POSIX 1003.1 *Standard Portable Operating Systems Interface for Computer Environments*, First Edition, IEEE, 1988.

[POSIX 93a] POSIX 1003.4/D14 *Standard Portable Operating System Interface for Computer Environments*, First Draft Edition, IEEE, 1993.

[POSIX 93b] POSIX 1003.4a/D7 *Standard Portable Operating System Interface, Real Time Thread Extension*, First Draft Edition, IEEE, 1993.

[Powell 91] Powell, D. *Delta-4 : A Generic Architecture for Dependable Distributed Computing*, Springer-Verlag, 1991.

[RAB 93] The RAID Advisory Board The RAIDBook: A Source Book for RAID Technology. *The RAID Advisory Board*, pp.81–90, Edition 1-1, November 1993.

[Ramamrithan et al 88] Ramamrithan, K., and Stankovic, J.A., Editors *Hard real-time systems*, IEEE Computer Society Press, 1988.

[Randell 75] Randell, B. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, pp.220–232, Vol.SE-1, No.2, June 1975.

[Randell et al 78] Randell, B., Lee, P., and Treleaven, P.C. Reliability Issues in Computing System Design. *ACM Computing Surveys*, pp.123–166, Vol.10, No.2, June 1978.

[Rashid 86a] Rashid, R.F. Threads of a new system. *UNIX Review*, pp.37–49, Vol.4, No.8, August 1986.

[Rashid 86b] Rashid, R.F. From RIG to Accent to Mach: The Evolution of a Network Operating System. *Proceedings of AFIPS 1986 Fall Joint Computer Conference*, pp.1128–1137, 1986.

[Reuter 80] Reuter, A. A Fast Transaction-Oriented Logging Scheme for Undo Recovery. *IEEE Transactions on Software Engineering*, pp.348–356, Vol.SE-6, No.7, July 1980.

[Rozier et al 88] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Léonard, P., Langlois, S., and Neuhauser, W. The Chorus Distributed Operating System. *Computing Systems-Usenix*, pp.305–370, Vo.1, No.4, 1988.

[Rudolph et al 84] RUDOLPH, L., AND SEGALL, Z. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. *Procseedings of 11th International Symposium on Computer Architecture*, pp.340–347, Ann Arbor, 1984.

[Sahner et al 95] SAHNER, R.A., TRIVEDI, K.S., AND PULIAFITO, A. *Performance and Reliability Analysis of Computer Systems*, Kluwer Academic Publishers, November 1995.

[Schlichting et al 83] SCHLICHTING, R.D., AND SCHNEIDER, F.B. Fail-Stop Processors: An Approach to Designing Fault- Tolerant Computing Systems. *ACM Transactions on Computer Systems*, August 1983.

[Schmid et al 82] SCHMID, M., ET AL Upper Exposure by Means of Abstraction Verification. *Proceedings of 12th International Symposium on Fault-Tolerant Computing Systems*, pp.237-244, St.Monica, June 22-24 1982.

[Schneider 87] SCHNEIDER, F.B. The Fail-Stop Processor Approach. In *Concurency Control and Reliability in Distributed Systems, Chapter 13*, pp.370–394, Barghava, 1987

[Schultze 88] SCHULTZE, M.E. Considerations in the Design of a RAID Prototype. *Technical Report UCB/CSD 88/448*, University of California, August 1988.

[Sha et al 87] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J.P. Priority inheritance protocols, an Approach to Real-Time Synchronization. *Technical Report CMU-CS-87-181*, Department of CS, ECE and Statistics, Carnegie Mellon University, November 1987.

[Siewiorek et al 92] SIEWIOREK, D.P., AND SWARZ, R.S. Reliable Computer Systems: Design and Evaluation. *Digital Press*, 1992.

[Singh et al 91] SINGH, J.P., WEBER, W., AND GUPTA, A. Splash : Stanford Parallel Applications for Shared-Memory. *Technical Report CSL-TR-91-469*, Computer Systems Laboratory, Stanford University, April 1991.

[Smith 82] SMITH, A.J. Cache Memories. *ACM Computing Surveys*, pp.473–530, Vol.14, No.3, September 1982.

[Spivey 92] SPIVEY, J.M. *The Z Notation - A Reference Manual*, 2nd Edition, Prentice Hall, 1992.

[Stonebraker 87] STONEBRAKER, M. The Design of the Postgres Storage System. *Proceedings of 13th International Conference on Very Large DataBases*, 1987.

[Strom et al 85] STROM, R.E., AND YEMINI, S. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, pp.204–226, Vol.3, No.3, 1985.

[Sweazey et al 86] SWEAZEY, P., AND SMITH, A.J. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. *Proceedings of 13th Annual International Symposium on Computer Architecture*, pp.414–423, ACM/IEEE, Tokyo, June 1986.

[Texas Instruments 94] Futurebus+ Interface Family: Protocol, Arbitration and Backplane Transceiver Logic, Data Manual. *Texas Instruments*, March 1994.

[Thacker et al 87] THACKER, C., AND STEWART, L. Firefly: A Multiprocessor Workstation. *Computer Architecture News*, pp.164–172, Vol.15, No.5, October 1987.

[Thacker et al 88] THACKER, C.P., STEWART, L.C., AND SATTERTHWAITE, E.H. Firefly : A Multiprocessor Workstation. *IEEE Transactions on Computers*, pp.909–920, Vol.37, No.8, August 1988.

[Thatte 91] THATTE, S.M. *United States Patent Specification No. US-A-5, 008, 786*, 1991.

[Thiebaut et al 92] THIEBAUT, D., ET AL Synthetic Traces for Trace-Driven Simulation of Cache Memories. *IEEE Transactions on Computers*, pp.388–410, Vol.41, No.4, April 1992.

[Webber et al 91] WEBBER, S., AND BEIRNE, J. The Stratus Architecture. In *Proceedings of 21st International Symposium on Fault-Tolerant Computing Systems*, June 1991.

[Wensley 81] WENSLEY, J.H. Fault-Tolerant Computers Ensure Reliable Industrial Controls. *Electronic Design*, Vol.29, No.13, 1981.

[Wensley et al 82] WENSLEY, J.H., AND HARCLERODE, C.S. Programmable Control of a Chemical Reactor Using a Fault Tolerant Computer. *IEEE Transactions on Industrial Electronics*, November 1982.

[Wood 85] WOOD, W.G. Recovery Control of Communicating Processes in Distributed Systems. In *Reliable Computer Systems*, Edited by Shrivastava, S.K., pp.448–484, Springer Verlag, 1985.

[Wilson 85] WILSON, D. The Stratus Computer System. In *Resilient Computer Systems* Edited by Anderson, T., pp.208–231, 1985.

[Wu et al 90] WU, K.L., FUCHS, W.K., AND PATEL, J.H. Error Recovery in Shared Memory Multiprocessors Using Private Caches. *IEEE Transactions on Parallel and Distributed Systems*, pp.231–240, Vol.1, No.2, April 1990.

[X/Open 91] X/OPEN Distributed Transaction Processing: The XA Specification. In *X/Open CAE Specification*, ISBN 1-872630-24-3, The X/Open Company Ltd., 1991.

[Yang et al 88] YANG, Q., AND BHUYAN, L.N. A Queuing Network Model for a Cache Coherence Protocol on Multiple-bus Multiprocessor. *Proceedings of Parallel Processing Conference* pp.130-137, August 1988.

Index

Abstract Execution, 72
Analytical Models, 33
    bus queue metrics, 39
    cache line states, 36
    invalidate protocol, 34
    performance metrics, 41
    protocol descriptions, 34
    update protocol, 35
Autobahn II, 99

Caches, 32
    TAGRAM cache for *PIX*, 159
    Berkeley protocol, 32, 71, 110
    blocking cache, 23
    cache for transputer, 153
    flushing timeout, 90
    influence on SM, 71
    recovery cache, 23, 70
    snooping by SM, 110
    stable memory as disk cache, 151
CARER, 23, 45, 48, 72
Checkpointing
    inter-bank, 155
    intra-bank, 153
    log-mode, 155
    switch-mode, 157
Corollary, 145, 187, 194
    base CPU, 145, 194
    binding threads to CPUs, 194
    EC-Bus, 145, 187, 194
    EISA Bus, 145, 187, 194
    OSF1/mk configuration file, 188
    symmetric CPU, 145, 194

Dependencies
    management, 75, 85
    tracking in SM, 67
    write read, 28
    write write, 28
Disk Arrays, 140
DT-Connect I, 149
DT-Connect II, 149
Dual Processing Unit (DPU), 116
    demonstrator, 122
    demonstrator software, 131
    error detection levels, 130
    evaluation of dependability, 134
    Futurebus+ interface, 129
    prototype, 122

EISA Bus, 145, 187, 194
Experimental Validation, 135

fail-silent, 116
fail-stop, 21, 116
FASST Architecture, 66
Fault Injection, 136
Futurebus+, 116, 129
    in stable memory, 99

Models
    analytical - see Analytical Models, 33
    queueing - see Queueing Models, 45

nMR
    in Stable Disk (SD), 150

Queueing Models, 45
    bridge function, 48
    CARER, 45
    degradable with recovery, 48
    dependability analysis, 47
    FASST recovery protocol, 45
    more than one node, 48
    non-degradable with recovery, 48
    QNAP2, 45
    SMPL, 45
    SMRC, 45
    without recovery, 48

RAID, 141
    levels, 141
    multiple ranks, 144
    RAID 0, 142
    RAID 1, 142
    RAID 2, 142
    RAID 3, 143
    RAID 4, 143
    RAID 5, 144
    the first prototype, 145
Recovery
    planned, 26
    recovery lines, 26

unplanned, 26

Recovery Protocol, 24, 45, 84

SM behaviour during recovery, 95

`Inform_p` primitive, 98

definitions, 24

group computation, 92

initiating processor, 91

locking, 98

model of computation, 24

other processors, 93

principles, 26

rollback due to processor failure, 95

rollback of all processors, 97

rollback of dependency group, 96

Secondary Storage, 140

Sequoia, 21, 23, 72

SMRC, 45, 48

Stable Disk (SD), 145

information flow, 149

integration into recovery protocol, 160

inter-bank checkpointing, 155

intra-bank checkpointing, 153

log-mode checkpointing, 155

nMR, 150

protection logic, 158

RAID controller, 147

stable memory, 151

switch-mode checkpointing, 157

VSBus, 149

Stable Memory (SM), 66, 84

atomic operations, 95

behaviour during recovery, 95

commit, 67

dependency management, 75, 85

dependency tracking, 67

influence of caches, 71

performance evaluation, 72

read command, 91

synchronization, 87

timeout protection, 89

use as disk cache, 151

write command, 91

Stable Memory Hardware, 99

Vector memory, 109

C012 link interface, 153

command and status registers (CSR), 109

copy-on-write, 100

dependency matrix memory, 109

expected performance, 111

Fast Serial Link (*FSL*), 99

fault tolerance issues, 111

for Stable Disk (SD), 151

Futurebus+, 99

information flow, 99

initialization phase, 110

snooping, 110

T800 Transputer, 153

transputer cache, 153

update memory, 110

Stratus, 21

T800 Transputer, 153

C012 link interface, 153

cache, 153

Transputer Development System, 153

Tandem

S2, 21

Tandem-16, 21

TMR, 21

VSBus, 149