USING LIGHTWEIGHT CHECKPOINT/RECOVERY TO IMPROVE THE AVAILABILITY AND DESIGNABILITY OF SHARED MEMORY MULTIPROCESSORS

by

Daniel J. Sorin

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN - MADISON

2002

Abstract

In this thesis, we address the issues of availability and designability for shared memory multiprocessors. While Moore's Law has provided architects with faster and more numerous transistors, it has correspondingly degraded availability and designability. Availability is increasingly difficult to achieve due to the combination of smaller devices and wires, along with more components in more aggressive designs. Designability, which we define as the difficulty of designing and verifying a computer system, has also suffered as a result of these same trends. As computer architects are given more transistors with which to work, system designs generally become much more complicated and more difficult to verify correct.

To address these downward trends in availability and designability, we propose using a lightweight checkpoint/recovery scheme called *SafetyNet*. *SafetyNet* is a hardware-only scheme that allows a shared memory multiprocessor to recover its system-wide state—including processor registers, caches, and memories—to a previous checkpoint. Thus, in the case of an error due to a device fault or a design fault, *SafetyNet* allows the system to recover to a pre-fault state and re-execute. In the case of transient faults and some permanent faults (i.e., those permanent faults that can be tolerated by reconfiguration), recovery and re-execution is sufficient to transparently tolerate the faults.

SafetyNet has three distinguishing features that enable it to provide error-free performance that is comparable to that of an unprotected system. First, it coordinates the system-wide checkpoints in logical time and leverages "logically atomic" cache coherence transactions. By using logical time, *SafetyNet* does not have to quiesce the system or exchange synchronization messages at each checkpoint. Second, *SafetyNet* uses an optimized logging scheme to reduce the amount of checkpoint state. Third, it pipelines checkpoint validation—the process of determining that a checkpoint is error-free and can be made the new recovery point—and keeps it entirely in the background.

We demonstrate that *SafetyNet* can be used in conjunction with a variety of existing error detection schemes to improve system availability. We also use *SafetyNet* to innovate in the areas of availability and designability. To improve availability, we leverage *SafetyNet's* ability to tolerate long error detection latencies. *SafetyNet* can tolerate latencies that are long enough to enable much stronger error detection techniques than are currently feasible. These techniques can use inter-node communication and system-wide invariant checking. To improve designability, we use *SafetyNet* to enable *speculatively correct designs*, as well as to tolerate certain classes of unintentional design faults. For rare and complicated system events, we demonstrate that we can fall back on *SafetyNet* (and treat these events as errors) instead of devoting

design time and verification effort towards handling them. Speculative correctness can simplify the design and/or enable otherwise infeasible design points.

We evaluate *SafetyNet* with full-system simulation and commercial workloads running on Solaris 8. Our results show that *SafetyNet* has negligible impact on error-free performance, while avoiding data corruption and system failures when errors are detected. We show that *SafetyNet* can provide this error recovery with reasonable storage costs (512-kbyte buffers at each processor and each memory) and with negligible additional cache bandwidth.

Acknowledgments

This thesis would not have made it to this point without the contributions of many people, and I wish to thank as many of them as I can remember. My parents and sister have been a wonderful source of support, inspiration, and encouragement throughout my education, and they deserve much credit for where I am today. Deborah, my lovely fiancee, is my inspiration, and she deserves all the thanks in the world, despite claiming not to understand what I do. I would not be who I am today without her, and I can only hope to make her as happy as she makes me.

The University of Wisconsin has been a wonderful place for me to pursue my Ph.D., and much of this is because of its outstanding faculty. My advisor, Prof. David Wood, has mentored me for the past six years, and I have learned how to perform research under his wise guidance. I thank David for his support and for forcing me, sometimes against my wishes at the time, to be a better researcher. Prof. Mark Hill, the other director of the Multifacet project, has been a joy to work with. Besides the many technical insights I gleaned from Mark, I thank him for his invaluable advice and for providing a terrific role model for how to be a researcher. Prof. Mary Vernon, with whom I worked early in my graduate career, led me through my first research. I thank Mary not only for teaching me how to perform and present research, but also for pushing me to the limits of my abilities (which, I might add, are still well short of hers). My collaboration with Prof. Anne Condon taught me how to think more formally and precisely, while simultaneously demonstrating that I better not switch areas into theory.

Other faculty at Wisconsin and beyond have contributed to where I am now. Prof. Derek Eager at the University of Saskatchewan has been a valuable collaborator, and I particularly enjoyed working with him, even when I got stranded in Saskatoon. I thank Profs. Guri Sohi, Mikko Lipasti, and Kewal Saluja at Wisconsin for their valuable insights into this thesis work. I thank Prof. Alvy Lebeck at Duke University for helping to convince me to go to Wisconsin, and I also thank Prof. John Board at Duke for inspiring me to pursue computer architecture in the first place.

The students at Wisconsin are the other reason that it has been such a wonderful place for me to pursue my Ph.D. First and foremost, Milo Martin has been the best possible collaborator and partner in crime. My research has benefited more from Milo than from anyone else, and I owe him many thanks for his help, insights, support, and leftover pizza. I would like to thank Amir Roth (now Prof. Roth) for countless discussions, some of which did not involve comparisons of our respective hometown sports teams. I thank Craig Zilles and Ravi Rajwar, as well, for numerous valuable discussions of our research. Ravi also

deserves many thanks for always being willing to play tennis on a moment's notice. I thank Manoj Plakal for his collaboration on more Lamport clock papers than we could have imagined. I thank Carl Mauer, my officemate, for putting up with me, particularly during ISCA seasons when a less saintly officemate would have had me evicted. Lastly, I would like to thank the other members of the Wisconsin Multifacet project, many of whose contributions have improved this research.

Research that depends on many compute-years of full-system simulation is not achieved without a lot of help. For supporting Simics and answering countless questions at all hours, I thank the industrious folks at Virtutech, particularly Peter Magnusson, Bengt Werner, and Andreas Moestedt. At Wisconsin, the Computer Systems Lab and the Condor project (especially Erik Paulson) enabled me to soak up all those compute-years. I also thank Erin Miller and Alicia Walley for expertly solving the numerous administrative issues I encountered. Lastly, I thank Intel Corporation for providing me with an Intel Graduate Fellowship and my very own laptop.

Table of Contents

Abstract i

Acknowledgments iii

Table of Contents v

List of Figures ix

List of Tables xi

Chapter 1 Introduction 1

1.1 A Case for Supporting Availability 3

1.2 A Case for Supporting Designability 4

1.3 Background Material 5

1.3.1 Availability 5

1.3.2 Designability 6

1.4 SafetyNet: Unifying the Support for Availability and Designability 7

1.5 Classification of Errors Due to Device and Design Faults 9

1.5.1 Four Aspects of Error Characterization 10

1.5.2 Classifying Errors in the Taxonomy 11

1.5.3 Hardware Faults Not Tolerated 13

1.6 Thesis Contributions 14

Chapter 2 SafetyNet: Abstraction and Implementation 15

2.1 SafetyNet Abstraction 15

2.1.1 Incremental Checkpointing Via Logging 17

2.1.2 Creating Consistent Checkpoints in Logical Time 17

2.1.3 Validating Checkpoints and Deallocating Checkpoint State 19

2.1.4 Recovering the System to a Consistent Global State 21

2.1.5 Input/Output Commit Problems 22

2.1.6 Other Classes of Coherence Protocols and Memory Models 23

2.1.7 Integrating SafetyNet with Other Levels of Checkpoint/Recovery 24
2.2 Implementing SafetyNet 25
2.2.1 System Model 26
2.2.2 Logical Time Base 28
2.2.3 Logging 30
2.2.4 Checkpoint Creation 32
2.2.5 Checkpoint Validation and Deallocation of Checkpoint State 33
2.2.6 System Recovery and Restart 35
2.2.7 Implementation Details 36
2.2.8 Summary of Implementation 38
2.3 SafetyNet Conclusions 39

Chapter 3 SafetyNet Evaluation 41
3.1 High-Level Performance Model 42
3.1.1 Error-Free Performance 42
3.1.2 Performance in Presence of Errors 44
3.2 Methodology 44
3.2.1 Simulation Infrastructure and Target System 45
3.2.2 Workloads 47
3.3 Experiments 49
3.3.1 Experiment 1: Error-Free Performance 49
3.3.2 Experiment 2: Dropped Messages 50
3.3.3 Experiment 3: Lost Switch 51
3.4 Sensitivity Analyses 51
3.4.1 Checkpoint Log Buffer Storage Cost 52
3.4.2 Checkpoint Interval Length 53
3.4.3 Register Checkpointing Latency 55

3.4.4 Sensitivity to the Rate of Soft Errors 56
3.4.5 Cache Bandwidth 57
3.5 Summary 58

Chapter 4 Availability 59
4.1 Traditional Hardware Error Detection Mechanisms 60
4.1.1 Interconnection Network Errors 60
4.1.2 Coherence Protocol Errors 61
4.1.3 Cache Hierarchy and Memory Errors 61
4.1.4 Processor Core Errors 62
4.1.5 SafetyNet Hardware Errors 62
4.1.6 Device Faults Not Tolerated with SafetyNet 63
4.2 Global Recovery versus Local Recovery 63
4.2.1 General Discussion of FER vs. Global BER 64
4.2.2 Interconnect Link Errors 65
4.2.3 Processor Errors 66
4.3 Innovations in Hardware Error Detection 66
4.3.1 Detecting Errors with Signature Analysis 67
4.3.2 Developing a Simplified Signature Analysis Example 69
4.3.3 Checking Message-Level Invariants with Signature Analysis 69
4.3.4 Checking Coherence-Level Invariants with Signature Analysis 72
4.4 Summary of Availability 74

Chapter 5 Designability 77
5.1 Errors due to Speculatively Correct Design 77
5.1.1 Simplifying Deadlock Avoidance in Interconnection Network Design 78
5.1.2 Enabling Adaptive Routing in the Interconnection Network 81
5.1.3 Avoiding Pathological Mis-speculation 85

5.2 Errors Due to Unintentional Design Faults 85
5.2.1 An Example in the Cache Coherence Protocol 87
5.2.2 General Properties 89
5.3 Summary of Designability 90

Chapter 6 Related Work 91
6.1 Availability 91
6.1.1 Hardware Backward Error Recovery 91
6.1.2 Software Backward Error Recovery 92
6.1.3 Message Passing Backward Error Recovery 93
6.1.4 (Hardware) Forward Error Recovery 94
6.2 Designability 95
6.3 Checkpoint/Recovery and Versioning of Data for Other Purposes 95
6.4 Using Logical Time to Coordinate Multiprocessor Systems 96

Chapter 7 Summary 99

References 101

Appendix A: Tabular Specification of SafetyNet Directory Protocol 111

List of Figures

1-1 SafetyNet abstraction 7
1-2 Example SafetyNet system implementation 9
2-1 SafetyNet abstraction 16
2-2 Example of checkpoint coordination 20
2-3 Checkpoint log buffer (CLB) structure 26
2-4 SN-Snooping system model 28
2-5 SN-Directory system model 28
2-6 Ensuring that logical time respects causality 30
2-7 Logging at the cache 31
2-8 Two-phase validation of checkpoint CPi 34
2-9 Two-phase recovery/restart 36
3-1 Performance comparison of SafetyNet with an unprotected system 50
3-2 Workload intensity (Apache workload) 52
3-3 Performance vs. CLB size 53
3-4 Performance as a function of checkpoint interval (512 kbyte CLBs) 54
3-5 Performance vs. CLB size for 500,000 cycle intervals 55
3-6 Performance vs. CLB size for 1 million cycle intervals 55
3-7 Performance as a function of register checkpointing latency 56
3-8 SafetyNet performance vs. soft error rate 57
3-9 Bandwidth vs. checkpoint interval (static web workload) 58
4-1 Rough comparison of BER vs. FER 64
5-1 Example of deadlock in interconnection network 79
5-2 Violating point-to-point order with adaptive routing 82
5-3 Performance of a system with adaptive routing 84
A-1 Protocol state machines and their incoming queues 112

List of Tables

1-1 Classification of illustrative errors . . . 12
2-1 Modifications to SafetyNet cache behavior . . . 37
3-1 Target system parameters . . . 45
3-2 Workload execution behavior . . . 49
4-1 Classification of illustrative errors due to device faults . . . 60
4-2 Coherence-level signature update function (SN-Snooping) . . . 73
5-1 Classification of illustrative errors due to speculatively correct design . . . 78
5-2 Classification of illustrative errors due to unintentional design faults . . . 86
A-1 SN-Directory - cache controller states . . . 113
A-2 SN-Directory - cache controller actions . . . 114
A-3 SN-Directory - cache controller events . . . 115
A-8 SN-Directory - directory controller states . . . 116
A-4 SN-Directory - cache controller transitions (part 1 of 4) . . . 117
A-5 SN-Directory - cache controller transitions (part 2 of 4) . . . 118
A-6 SN-Directory - cache controller transitions (part 3 of 4) . . . 119
A-7 SN-Directory - cache controller transitions (part 4 of 4) . . . 120
A-9 SN-Directory - directory controller actions . . . 121
A-10 SN-Directory - directory controller events . . . 122
A-11 SN-Directory - directory controller transitions . . . 123
A-12 SN-Directory - network interface actions . . . 124
A-13 SN-Directory - network interface events . . . 125
A-14 SN-Directory - network interface transitions . . . 125

Chapter 1

Introduction

Computers are becoming increasingly important in our daily lives. We rely upon computer services, such as electronic commerce and information delivery. These services are based upon software infrastructure, such as database management systems and web hosting software. This software infrastructure, in turn, depends upon the hardware infrastructure provided by computer systems. These machines, known as commercial servers, have become an integral part of society's infrastructure. These commercial servers are often systems with multiple processors that share a single memory address space. Shared memory multiprocessors provide the performance necessary for running commercial applications, especially for workloads that exhibit abundant concurrency, such as databases and web servers.

Computer system performance, including that of shared memory multiprocessors, has harnessed the exponential trend described by Moore's Law and complemented by architectural advances. Performance has traditionally been the focus of most research in computer architecture, and increasing performance has enabled qualitative improvements in computing. But beyond performance, there are other features that contribute to the quality of a commercial server. This thesis addresses two other system aspects that contribute towards system quality: *availability* and *designability.* Unfortunately, as we will discuss, Moore's Law presents increasing challenges for both of these issues.

Availability is the probability that a system is functioning correctly at a given time. Availability is often confused with *reliability,* which is the probability that a system that is functioning correctly at a given time will continue to function correctly for a specified amount of time afterwards. A system may function incorrectly if a fault ("a physical defect, imperfection, or flaw" [80]) manifests itself as an error ("a deviation from accuracy or correctness" [80]). Availability is a function of both the frequencies of faults and the system's ability to tolerate them—either by masking them or recovering from them—when they manifest themselves as errors. A highly available system must thus avoid and/or tolerate faults. Avoidance is difficult, though, because the smaller and more numerous transistors enabled by Moore's Law degrade availability by increasing fault frequencies. Patterson argued recently that faults are a fact of life and that designers must learn to deal with them [72]. In Section 1.1, we further argue why architects should develop systems that provide improved availability by tolerating increasing numbers of faults.

We define designability as the ease with which a system can be designed and verified. As with availability, Moore's Law causes designability to suffer. Having more transistors with which to work enables architects to design more complicated systems that are more difficult to design and verify. Designability is important because it relates directly to cost, for two broad reasons. First, if it takes longer to design a system and longer to verify that the system is correct, this increase in time translates to lost performance. Since Moore's Law has provided exponential increases in transistors and, in turn, performance over time, unforeseen additional time to market incurs an exponential loss in performance. For example, an eighteen month product slip is roughly a factor of two in performance that has been lost. Second, systems that target worst-case scenarios, in an effort to simplify design and verification, are often far costlier, in terms of resources and performance, than designs that could target the common case. For both of these reasons, we would like to be able to design systems more quickly and more efficiently. In Section 1.2, we present a case for improving system designability.

There exists a large body of prior research in available systems and some limited current research in designability. In Section 1.3, we present a brief overview of this existing work, in order to illustrate its achievements and limitations, thus motivating our research in these areas. We present a full treatment of the related work in Chapter 6.

To address the issues of availability and designability, this thesis presents a system-wide checkpoint/recovery scheme called *SafetyNet*. *SafetyNet*, described briefly in Section 1.4 and in depth in Chapter 2, unifies the support for availability and designability by recovering the system to a pre-fault state in the case of errors due to both device faults and design faults. The recovery point is a globally consistent checkpoint of the system state, including the memory system and cache coherence protocol, enabling the system to seamlessly resume execution after recovery. We described *SafetyNet* in previous work [100], but that research only addressed the use of *SafetyNet* for improving availability with conventional error detection techniques. This thesis not only explores this area in greater depth, but it also develops innovative error detection mechanisms and addresses the issue of system designability.

To illustrate the similarities between availability and designability, we present a framework for classifying errors due to device and design faults in Section 1.5. We classify errors based on four aspects—fault, detection, recoverability with checkpoint/recovery, and resumability—and we classify several illustrative examples to show the utility of the framework. This framework will be used throughout the thesis, when discussing particular error models.

The cornerstone of our philosophy is that we want to allocate our resources—transistors, performance, design effort, verification time—towards the common case of normal, error-free operation. In the rare case

of an error, be it due to a transient device fault or a design fault, we will fall back on our checkpoint/recovery mechanism. Moreover, error detection latency should be hidden, since finding a error is not the common case. This philosophy reflects both architectural common sense as well as the philosophy of optimistic algorithms, including optimistic distributed protocols [74] and optimistic concurrency [57]. Optimistic algorithms speed up the common case (or do not slow it down), while sacrificing some performance in the uncommon case that the optimistic assumption fails. In the context of availability and designability, the optimistic assumption is that the execution is error-free.

The primary contributions of this thesis, detailed in Section 1.6, are the improved availability and designability that are enabled by *SafetyNet*.

1.1 A Case for Supporting Availability

Availability becomes increasingly important as computer services are integrated more tightly into society's infrastructure. This is particularly true for the shared-memory multiprocessor servers that run the application services and database management systems (DBMSs) that must robustly manage business data. However, unless architectural steps are taken, availability will decrease over time as implementations use a larger number of increasingly unreliable components in search of higher performance [21, 42, 109]. The high frequencies and small circuit dimensions of future systems will increase their susceptibility to both transient and permanent faults. For example, higher frequencies exacerbate crosstalk [5, 14] and supply voltage noise [93], and smaller devices and wires suffer more from electromigration [114] and alpha particle disruptions [89, 123]. Moreover, while DRAM cells have long been susceptible to radiation-induced faults, now SRAM cells and combinational logic are also susceptible [52, 94].

Decades of research in fault-tolerant systems suggest a path toward addressing this problem. Mission-critical systems routinely employ redundant processors, memories, and interconnects (e.g., triple-modular redundancy [53] or pair-and-spare [117]) to tolerate a broad class of faults. However, for many applications, the highly competitive commercial market will seek lighter-weight solutions. For example, RAID level 5 [73] has been deployed widely because its overhead is 1/Nth (for N data disks) rather than the 100% overhead for mirroring. Commercial servers aim for high availability but will accept occasional crashes to improve cost/performance. Software-visible techniques—including database logging and clustering—help preserve data integrity in these cases. However, availability is compromised, because recovery from a crash takes hours, and "5 9s" of availability (i.e., 99.999% availability) translates to only five minutes of downtime per year.

Current servers employ a range of hardware mechanisms to improve availability. Error correcting codes (ECC), interconnection network link-level retry [36], RAID [73], and duplicate ALUs with processor retry [102] target specific, localized error models such as transient bit flips on memory, links, or ALUs. Computer architects seeking system-wide coverage must integrate a patchwork of localized error detection and recovery schemes.

In this thesis, we propose using the *SafetyNet* checkpoint/recovery scheme to provide a unified, lightweight mechanism that provides end-to-end recovery from a broad class of transient and permanent errors due to device faults (which we refer to as *device errors*). This recovery mechanism can be combined with a wide range of error detection mechanisms, including strong error detection codes (e.g., CRCs), redundant processors and ALUs [36, 102], redundant threads [91], and system-level state checkers [16]. By largely decoupling recovery from detection, our approach allows a range of implementations with varying cost-performance. By providing a unified mechanism that can tolerate an increasingly important class of transient and permanent errors, we hope to encourage pervasive use of *SafetyNet* in commercial servers.

1.2 A Case for Supporting Designability

Shared memory multiprocessors are complicated systems that are difficult to design for a variety of reasons. Most significantly, interprocessor communication and cache coherence protocols are prone to timing races that result in rarely exercised corner cases. Not only is it a challenge to design these systems correctly, it is even more difficult to do so without wasting hardware resources. For example, it is often difficult to design a cache coherence protocol that is not wasteful of buffering in the interconnect and coherence controllers, and it is tempting to employ worst-case buffering rather than to add complexity to reduce this requirement. However, there are many design issues, including buffer sizing, for which the worst-case requirements may be far worse than the requirements for the common case.

In this thesis, we propose using *SafetyNet* to improve designability. With such a checkpoint/recovery mechanism, when a design fault manifests itself as an error (which we refer to as a *design error*), we can recover the system state to a pre-error checkpoint. Our proposal relies on two assumptions. First, we must be able to detect, in a timely fashion, that an error due to a design fault has occurred. Second, we must be able to ensure that, after recovery, the system will not immediately encounter this design fault again, thus leading to livelock. In Chapter 4, we will discuss our fault model, error model, and error detection schemes, as well as how to avoid livelock. We seek to leverage the *SafetyNet* mechanism that was proposed for availability to provide the additional benefit of improved designability.

There are two classes of situations in which we will fall back on our checkpoint/recovery mechanism. The first class of situations are errors due to (unintentional) design faults. Since design faults are, by definition, unintentional, it can be difficult to devise detection schemes to catch all of them. As we will discuss in Section 5.2, *SafetyNet* can tolerate some unintentional design faults, because they manifest themselves like device faults. While *SafetyNet* cannot be relied upon to tolerate all design faults, we can improve our odds of both detecting the errors they cause and enabling the resumption of execution after recovery. We improve our chances of detection by employing more comprehensive (device) error detection. We improve our chances of being able to resume execution—and not livelock due to immediately re-encountering the error—by designing more adaptable and flexible systems that can change the execution path after recovery.

The second, more profitable, class of situations in which we recover the system state are errors due to *speculatively correct design*. Speculatively correct design can be useful in situations in which the requirements to handle an infrequent event are far worse than the common case. Designers like to focus their resources on the common case, but resources often get allocated to uncommon cases, since not handling these uncommon cases can cause system failure. Errors due to mis-speculation (i.e., speculation in the sense that we optimistically predicted that a certain edge case would not get exercised) are much easier to detect, since the designer knows exactly where they are. We will present two speculatively correct designs in Section 5.1. In one example, we enable adaptive routing in an interconnection network that must provide (the illusion of) point-to-point ordering. When adaptive routing leads to violations of point-to-point ordering *that impact correctness* (most re-orderings do not matter), the system falls back on *SafetyNet* and resumes execution with the adaptive routing disabled until after execution has progressed beyond the point of the error. With *SafetyNet* support, we can gain the advantages of adaptive routing while hiding the rare occasions in which our optimistic assumption is violated.

1.3 Background Material

In this section, we briefly address related work in the areas of availability and designability, in order to provide a context in which to discuss the contributions of this thesis. We provide a full treatment of related work in Chapter 6.

1.3.1 Availability

Many availability schemes have been developed, but they have achieved availability at the cost of either degraded performance or significant additional hardware. These schemes can be classified into two categories: *forward error recovery* (FER) and *backward error recovery* (BER). Forward error recovery schemes

use redundant hardware, such as triple modular redundancy (TMR) to mask errors while execution continues running forward. IBM mainframes [96] are a prominent example of FER systems that achieve high availability. The primary cost of FER schemes is the redundant hardware. While the cost of TMR or other FER schemes may be acceptable for mission-critical systems, commercial servers seek better cost/performance.

Backward error recovery schemes, including *SafetyNet*, use a combination of checkpointing and/or logging to enable the system to recover to a pre-error state, the *recovery point*, in the case of an error. Checkpointing schemes periodically save the state of the system and recover to the recovery point in the case of an error. Logging schemes log changes to the system state and then undo the logs if an error is detected. BER schemes have been developed in both hardware and software, and the primary cost of these schemes is a degradation in system performance. Before *SafetyNet*, coordination of checkpoints/logs across the system and the checkpointing of system state have tended to be inefficient. While performance may not be crucial for mission-critical systems, it is an important factor for commercial systems and, as such, they have tended not to employ hardware BER. Software BER is still used in applications such as database management systems, but the latency of software checkpointing determines that it must be infrequent. Moreover, the latency of software recovery determines that errors cannot be a frequent occurrence, as may be the case in the near future. Even an error rate as low as one error per trillion cycles is equivalent to an error every seventeen minutes at a clock rate of 1 GHz or an error every ten seconds for a 100 GHz clock.

1.3.2 Designability

Little research exists in the area of designability, but recent research in dynamic verification suggests that this topic is attracting interest. DIVA [6] is the primary example of a designability scheme. DIVA adds a simple, provably correct checker processor at the retirement stage of an aggressive processor that is not formally verifiable. The checker processor dynamically verifies that the aggressive processor is producing correct results. In the case that the aggressive processor exhibits an error (due to a design fault or a device fault), the checker processor masks this error at the cost of degraded performance. While DIVA is an excellent approach for processor designability, it does not address multiprocessor designability. The complexity of everything beyond the processors—including the interconnection network and cache coherence protocol—is increasing, and recent research in dynamic verification of cache coherence protocols [16] incurs a large hardware cost and an appreciable degradation in performance.

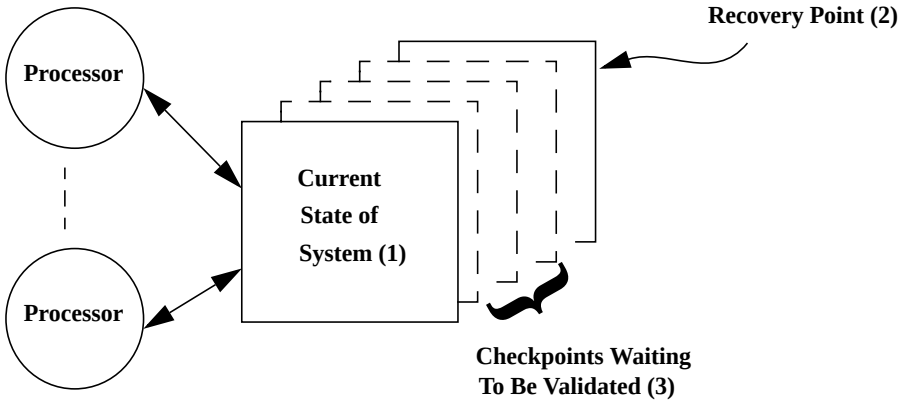

FIGURE 1-1. *SafetyNet* abstraction

In *SafetyNet*, (1) processors operate on the current state of the system, (2) the system can recover to the recovery point if an error is detected, and (3) some number of non-current checkpoints can be pending validation.

1.4 *SafetyNet*: Unifying the Support for Availability and Designability

This thesis improves availability and designability with a lightweight global checkpoint/recovery scheme called *SafetyNet.* As illustrated in Figure 1-1, *SafetyNet* periodically creates system-wide (logical) checkpoints. *SafetyNet* checkpoints can span thousands or even millions of execution cycles, permitting powerful detection mechanisms with long latencies. If an error is detected, all processors, caches, and memories revert to (and resume execution from) a consistent system-wide checkpoint state, the *recovery point*. *SafetyNet* is a hardware-only scheme that requires no changes to any software or the instruction set. Moreover, *SafetyNet* has limited impact on the processor, coherence protocol, and input/output (I/O) subsystem design.

SafetyNet's basic approach is to incrementally checkpoint system state by using logging. In between checkpoints, *SafetyNet* (logically) logs all changes to the architected state by saving the before image at each change.

There exist three main challenges for a lightweight checkpoint/recovery scheme that employs logging.

- Naively saving previous values before every register update, cache write, and coherence response would require a prohibitive amount of storage (i.e., multiple megabytes).

- All processors, caches, and memories in a shared-memory multiprocessor must recover to a consistent point. For example, recovery must ensure that all nodes agree on the ownership and data values of each memory block.
- *SafetyNet* must determine when it is safe to advance the recovery point (i.e., validate a new checkpoint), without degrading performance to wait for slow error detection mechanisms. The bottleneck is the slowest error detection mechanism, which is likely to be a timeout on a coherence request. A coherence timeout would likely be set to elapse after the latency of a couple traversals of the interconnection network plus some slack for worst-case contention.

SafetyNet efficiently meets these challenges. By doing so, *SafetyNet* checkpoint/recovery achieves high performance while maintaining a low hardware cost. There are three keys to *SafetyNet*'s efficiency.

- *Optimized logging*: Logging is reduced by checkpointing at a coarse granularity (e.g., 100,000 cycles). Only the first change to a piece of architectural state—register, memory block, or coherence permission—within a checkpoint interval requires a log entry, reducing the log overhead by one or two orders of magnitude.
- *Logical time checkpoint coordination*: *SafetyNet* efficiently coordinates checkpoint creation using *global logical time* and *logically atomic coherence transactions*, ensuring a consistent recovery point.
- *Pipelined validation*: Checkpoint validation is pipelined and overlapped with normal execution. Pipelining validation allows *SafetyNet* to tolerate long latency error detection mechanisms in the background.

We develop two *SafetyNet* implementations—one for a system with broadcast snooping cache coherence and one for a system with directory coherence—that minimize runtime overheads for actions in the common case of fault-free execution, including memory operations and coherence transactions. Figure 1-2 depicts, for a directory system, the hardware used to hold logged state—register checkpoint buffers and Checkpoint Log Buffers (CLBs)—that is added to processor-memory nodes in the directory system. Register checkpoints, CLBs, caches, and memories are deemed "stable storage" and protected by ECC. As currently defined, *SafetyNet* cannot recover from uncorrectable errors to these structures, which may encourage the use of stronger ECC codes [28] or fault tolerance schemes that provide redundancy for these structures [82]. Future work could address this class of faults, including processor-cache chip kills, but solutions will necessarily trade some performance to provide availability in this case.

SafetyNet is a recovery mechanism that is largely decoupled from any specific error detection mechanisms. *SafetyNet* reduces the problem of fault tolerance to the simpler problem of error detection. In Section 1.5, we present a framework for classifying errors due to both device and design faults, but we postpone discus-

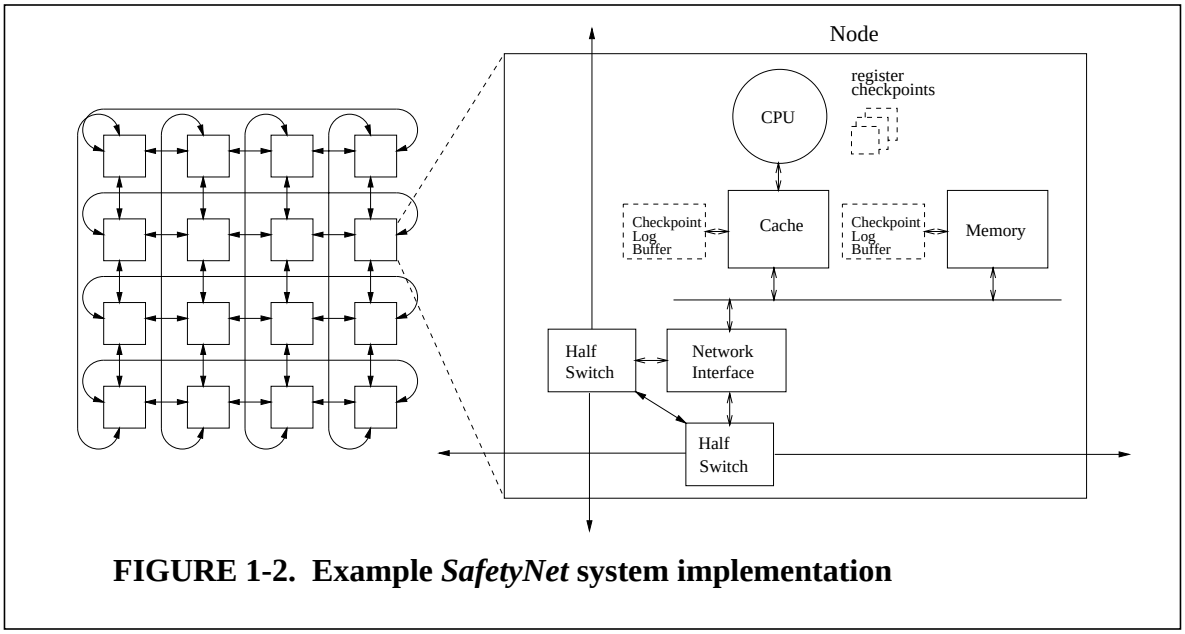

FIGURE 1-2. Example *SafetyNet* system implementation

sion of our specific fault models, error models, and error detection schemes until Chapter 4 and Chapter 5, when we describe the wide variety of faults and error detection mechanisms compatible with *SafetyNet*. Like most prior work, we focus on tolerating all single faults plus coverage for many but not all double faults.

In Chapter 3, we evaluate an implementation of *SafetyNet* with full-system simulations and commercial workloads. Our results show that, in the common case of error-free execution, *SafetyNet* does not increase execution time (relative to an unprotected system) by a statistically significant amount. Moreover, *SafetyNet* continues to run after the injection of faults. Recovery time is reduced from a system crash/reboot to a performance "speed bump" of less than one millisecond. We also show that 512-kbyte CLBs are large enough, for our commercial workloads, to tolerate error detection mechanisms with over 100,000 cycles of latency.

1.5 Classification of Errors Due to Device and Design Faults

The difference between *SafetyNet* support of availability and designability is equivalent to the difference between errors due to device faults and errors due to design faults. When designers use checkpoint/recovery to improve availability, they target errors (e.g., a bit flip) due to device faults (e.g., electrical crosstalk). Designers specify a particular fault model and corresponding error model, and then they design error detection schemes accordingly. The fault and error models for designability differ, but our designability approach leverages the same checkpoint/recovery mechanism to recover from design errors.

In this section, we classify hardware errors based on several aspects. We first define the aspects and then classify some illustrative error examples within the *error space*. We then discuss faults that are not currently tolerated by *SafetyNet*, why they are not tolerated, and how they might be tolerated in future work.

1.5.1 Four Aspects of Error Characterization

We characterize hardware errors based on four aspects: fault, detection, *SafetyNet* recoverability, and *resumability* mechanisms. Resumability mechanisms allow for the resumption of execution after recovery. We define a fault to be *tolerable* if it is both recoverable and resumable. We now discuss each of these aspects in turn.

Fault. An error can be caused by any number of faults. For example, a transient bit flip on a link in the interconnection network might be caused by crosstalk with a neighboring wire. Or, to give an example of a design error, a system deadlock might be caused by speculatively underdesigning the buffering in the interconnection network, as will be explained in Section 5.1.1.

Detection. This aspect of the error space specifies how an error can be detected (in a reasonable amount of time). For example, a bit flip on a link can be detected by an error detecting code (EDC). For the example error of a message lost in a dead switch in the interconnection network (ICN), the error can be detected by a time-out at the requestor.

A system does not detect a fault; rather, it detects the error that is the manifestation of the fault. Thus, detecting that an error occurred is not necessarily sufficient for diagnosing the fault. For example, a time-out on a coherence request detects that a response was not received, but it does not diagnose why the response was not received (e.g., dead switch in the interconnection network). Diagnosis is important if the fault requires that some action be taken after it occurs (e.g., reconfiguring the interconnect to route around a dead switch). Diagnosis can be performed in two stages. The first time an error is detected, the system assumes that the fault is transient. If the error is detected repeatedly, the system then invokes diagnosis mechanisms in hardware and/or software. For example, multiple time-outs on coherence requests could invoke a diagnosis mechanism in the interconnection network. Such a mechanism could force each switch to ping its neighbors and thus enable the interconnect to determine if a switch is dead.

***SafetyNet* Recoverability.** This aspect specifies whether *SafetyNet* can be used to recover from the fault, assuming we can detect its resultant error in the first place. *SafetyNet* can recover from a wide variety of detected faults by recovering to a pre-fault state. Moreover, *SafetyNet* hides the latency of error detection,

enabling stronger error detection mechanisms that would otherwise negatively impact performance. In Section 1.5.3, we will discuss unrecoverable/unresumable faults and how they might be made tolerable in future work.

Resumability Mechanisms. If *SafetyNet* can be used to recover from an error, it must also guarantee that execution can resume after the recovery, if this error is to be tolerable. For example, if resuming execution will lead immediately back to the error, then livelock is possible. This aspect of the error space specifies what, if any, techniques are needed to resume execution after recovering from the error. Such techniques include reconfiguration (e.g., to route around a dead switch in the interconnection network) and "slow-start" execution after recovery (e.g., to avoid the same timing race that manifested a fault due to speculatively correct design). For some errors, resumption of execution may require software assistance (e.g., if reconfiguring the routing in the interconnect cannot be performed entirely in hardware).

1.5.2 Classifying Errors in the Taxonomy

Using the four aspects described in Section 1.5.1, we will now characterize some illustrative errors, including errors due to both device and design faults. A tabular classification of them is shown in Table 1-1. Errors that are not currently tolerated by *SafetyNet* are shaded in the table.

Errors due to device faults. The first three errors in Table 1-1 are examples of errors due to device faults. The three examples are: dead switch in the interconnection network (ICN), dropped coherence message, and processor-cache chipkill. The unifying thread for these errors is their causes, which are all device faults, although one is transient and two are permanent. All of these errors are detectable and all but the processor-cache chipkill are recoverable. Resumability mechanisms depend on the fault model, although it is instructive to note that a system can resume execution after all errors due to transient device faults (not just the example here) without using any special mechanism.

Errors due to speculatively correct design faults. The next two errors in Table 1-1 are examples of errors due to speculatively correct design faults, and these two examples will be elaborated in Chapter 5. These errors share more in common than the other two categories. First, they all derive from the same type of "fault," which is speculative correctness. Second, they are all detectable (and easily diagnosed, as will be discussed later) and recoverable. Third, each of them requires a resumability mechanism for avoiding livelock after recovery. A speculatively correct design that cannot ensure livelock avoidance is not correct and cannot be employed.

TABLE 1-1. Classification of illustrative errors[a]

	Error	Fault	Detection	Recoverable with SafetyNet	Resumability Mechanism
errors due to device faults	dead switch in ICN	hard device fault	timeout on request	yes	reconfiguration
	dropped coherence message	soft device fault	timeout on request	yes	none needed
	proc-cache chipkill	hard device fault	watchdog timer	no	not available
errors due to speculatively correct design	deadlock due to insufficient buffering in ICN (Section 5.1.1)	speculative underdesign	timeout on request	yes	slow-start execution after recovery
	out of order message arrivals on “in-order” ICN (Section 5.1.2)	speculative use of adaptive routing	invalid transition in protocol engine	yes	disable adaptive routing during re-execution
errors due to unintentional design faults	unspecified edge case in coherence protocol (Section 5.2.1)	unintentional design fault	invalid state in protocol engine	yes	slow-start execution after recovery
	Intel’s FDIV bug [13]	unintentional design fault	self-checking program	yes	software FP routine

a. We shade the faults that *SafetyNet* cannot tolerate either at all or without software support.

Errors due to unintentional design faults. The last two errors in Table 1-1 are examples of errors due to unintentional design faults. The two examples are: an unspecified edge case in the cache coherence protocol and Intel’s FDIV bug [13]. The FDIV bug would not currently be detected in hardware, but it could be detected by a self-checking program [12]. If it was detected, it would be recoverable. However, execution could not resume without software intervention after recovery, since the execution would lead straight back to this fault and thus livelock. A software routine to perform floating point division would solve this problem.

1.5.3 Hardware Faults Not Tolerated

Certain hardware faults are not currently tolerable using *SafetyNet*. Faults can be unrecoverable and/or unresumable for one of several reasons, which we will now discuss. Recoverability, even without resumability, is still useful in that it avoids the loss or corruption of data. After recovery, the system could trap to software to gracefully exit.

Resultant error is undetected. A checkpoint/recovery scheme is only as good as its fault model and associated error detection schemes. For example, using parity detects all single-bit errors, but it will not detect double-bit errors. Thus, if faults that cause double-bit errors are to be added to the fault model, parity is not a sufficient detection mechanism. The system validates a checkpoint after all nodes have agreed that it is error-free according to their error detection mechanisms. Thus, an undetected error would be included in the validated state of the recovery point. Once this has occurred, system state is corrupted. The execution may still execute correctly, if this erroneous data is not used again (i.e., the erroneous data is dead), but this is good fortune rather than good planning. Note also that an error that is detected after the erroneous state has been validated is equivalent to an undetected error.

Fault corrupts recovery point state. Certain faults can corrupt state associated with the recovery point, which violates an integral *SafetyNet* assumption. *SafetyNet* assumes that its recovery point state is protected. Recovery point state includes processor register checkpoints, caches, checkpoint log buffers (CLBs), and memory/directory state. This state can be protected from many soft faults with error correcting codes (ECC). However, an uncorrectable soft fault or a hard fault that corrupted any of this state could be unrecoverable. An important area of future work is extending *SafetyNet* to handle these harder fault models. For example, a hard fault (e.g., a short circuit) that results in a processor-cache chipkill would destroy all of the state on the chip and partition that node’s memory banks from the rest of the system.

Cannot resume execution after recovery. Certain faults that are recoverable with *SafetyNet* do not permit resumption of execution after the recovery. For example, an fault that partitioned the system would not permit execution to resume after recovery. Our hard fault model includes faults that cause the loss of a half-switch, but not the loss of a whole switch (i.e., the switch was not split into two half-switches). Losing a whole switch causes the switch’s node to be partitioned from the rest of the system. *SafetyNet* can provide a recovery that preserves system data without any corruptions, but intervention is required before execution can resume. In this example, resumption requires manual intervention to replace a faulty component.

Other faults are recoverable with *SafetyNet* but are unresumable because there is no way to avoid livelock after recovery. For example, an unintentional design fault in the processor core that affected a specific instruction in a specific circumstance (e.g., Intel's FDIV bug [13]) could be tolerated by *SafetyNet* (assuming its resultant error was detected) without corruption of system data. However, resuming execution would lead immediately back to that fault and result in livelock. In this case, the system would need to invoke a higher level recovery mechanism, likely a software routine, to handle the problem. If a speculatively correct design fault falls into this category and it occurs frequently, then the system designer failed in her use of speculative correctness.

1.6 Thesis Contributions

This thesis makes the following contributions:

- We develop *SafetyNet*, a globally-consistent, hardware-only checkpoint/recovery scheme for shared memory multiprocessors. (Chapter 2)
- We evaluate *SafetyNet*'s performance and cost using full-system simulation and commercial workloads, and we compare it to an unprotected system. (Chapter 3)
- We show how to use *SafetyNet* to improve system availability. In the process, we innovate stronger error detection techniques, based on system-wide signature analysis, that are enabled by *SafetyNet*'s tolerance of detection latency. (Chapter 4)
- We show how to use *SafetyNet* to improve system designability. In the process, we describe potential avenues for speculatively correct design, and we show how to tolerate a certain class of unintentional design faults. (Chapter 5)

In Chapter 6, we discuss related research—in availability, designability, and logical time schemes—and compare *SafetyNet* to this work. Chapter 7 concludes this thesis and outlines some potential areas of future work.

Chapter 2

SafetyNet: Abstraction and Implementation

In this chapter, we develop *SafetyNet*, a hardware-only mechanism for globally consistent checkpoint/recovery of shared memory multiprocessors.[1] *SafetyNet* allows for the recovery of the global system state, in the case that an error occurs and is detected. We begin in Section 2.1 by describing an abstraction of the *SafetyNet* interface, and then we develop one specific implementation of *SafetyNet* in Section 2.2.

2.1 *SafetyNet* Abstraction

This section presents a high-level overview of the *SafetyNet* abstraction that is illustrated in Figure 2-1 (identical to Figure 1-1). The purpose of *SafetyNet* is to allow the system to recover its state to a consistent previous checkpoint, where a checkpoint includes all state necessary to resume execution after recovery: processor registers, memory values, and cache coherence permissions. While the processors are interacting with the active state of the system, *SafetyNet* is periodically taking system-wide checkpoints. The checkpoint most recently validated as being error-free is the system's *recovery point*, i.e., the checkpoint to which the system recovers in the case that an error is detected. Between the recovery point checkpoint and the active state of the system, some number of checkpoints may be pending validation.

Three challenges for logging schemes were raised in Chapter 1, and *SafetyNet* addresses all three. First, naively saving previous values before every register update, cache write, and coherence response would require a prohibitive amount of storage. *SafetyNet* addresses this challenge by exploiting a coarse checkpoint granularity to reduce the amount of logging (Section 2.1.1). Second, all processors, caches, and memories in a shared-memory multiprocessor must be able to recover to a consistent point. For example, recovery must ensure that all nodes agree on the ownership and data values of each memory block. To efficiently solve this problem, *SafetyNet* creates consistent global checkpoints in logical time (Section 2.1.2) such that all processors and memories can recover to a consistent recovery point upon error detection. Third, *SafetyNet* must determine when it is safe to advance the recovery point (i.e., validate a new check-

1. Checkpoint/recovery is performed entirely in hardware, although system reconfiguration between recovery and resumption (e.g., reconstructing interconnect routing to avoid a dead switch) of execution might require software assistance.

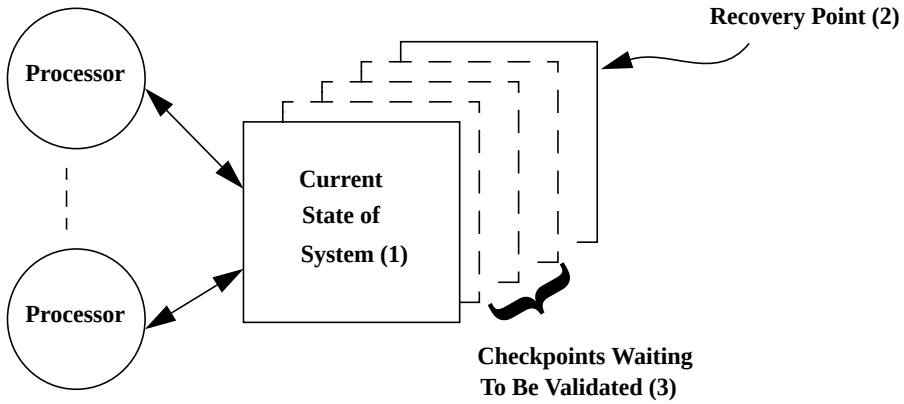

FIGURE 2-1. SafetyNet abstraction

In *SafetyNet*, (1) processors operate on the current state of the system, (2) the system can recover to the recovery point if an error is detected, and (3) some number of non-current checkpoints can be pending validation.

point), without degrading performance to wait for slow error detection mechanisms. To achieve this goal, *SafetyNet* enables pipelined checkpoint validation that is off the critical path and hides the latencies of error detection mechanisms (Section 2.1.3).

Beyond the three key features of *SafetyNet*, we discuss several other important issues. We describe the recovery process and how in-flight coherence transactions are handled (Section 2.1.4). We also discuss how *SafetyNet* adopts standard solutions for interacting with input/output devices (Section 2.1.5), how *SafetyNet* could be implemented with different cache coherence protocols and different memory consistency models (Section 2.1.6), and how *SafetyNet* interacts with other levels of checkpoint/recovery in computer systems (Section 2.1.7).

In the rest of this section, we will assume that the system implements cache coherence with a directory protocol and that the system supports a sequentially consistent memory model. In general, *SafetyNet* has only a small impact on the underlying cache coherence protocol, and *SafetyNet* does not affect the implementation of a sequentially consistent system. In situations in which *SafetyNet* affects the cache coherence protocol, we will highlight the impact and explain why modifications are necessary.

2.1.1 Incremental Checkpointing Via Logging

Logically, *SafetyNet* checkpoints contain a complete copy of the system's architectural state, which includes processor, cache, memory, and coherence state. *SafetyNet* explicitly checkpoints a processor's state by saving a copy of the processor's architected registers. Checkpointing of processor register state can be done in many ways, including shadow register copies or writing the registers into the cache, and we will discuss implementations of register checkpointing in Section 2.2. We only assume that the processors implement precise interrupts [97], since this assumption simplifies the checkpointing of architected state and all current processors maintain precise interrupts.

Explicitly checkpointing memory state, including cache and coherence state, would be inefficient. Instead, *SafetyNet* incrementally checkpoints memory state by logging the previous values of memory blocks and coherence permissions. Conceptually, cache controllers and memory controllers log every change to the memory/coherence state (i.e., save the *old* copy of the block) whenever an *update-action* (i.e., a store or a transfer of ownership) might have to be undone. To reduce storage and bandwidth requirements at the caches, where storage and bandwidth are more expensive than at memory, *SafetyNet* cache controllers only log a block on its first such update-action per checkpoint interval. By combining this optimization with coarse checkpoint intervals (e.g., 100,000 cycles), *SafetyNet* significantly reduces logging overhead (evaluated in Chapter 3). Implementations of logging will be discussed in Section 2.2.

2.1.2 Creating Consistent Checkpoints in Logical Time

All of the components (processors, cache controllers, and memory controllers) coordinate their local checkpoints, so that the collection of local checkpoints represents a consistent global recovery point. A consistent checkpoint is necessary for memory values and coherence permissions. Without consistency, for example, recovery could revert the system to a checkpoint state in which two nodes both believe that they own the same block. For a checkpoint to be consistent, all nodes must agree which coherence transactions occurred before the checkpoint and which occurred after it. Coordinated system-wide checkpointing has two advantages over independent checkpointing. First, it avoids the problem of cascading rollbacks [31], whereby recovering one node leads to recoveries on other nodes that cascade backwards in time until a consistent checkpoint line can be determined. Second, it eliminates an output commit problem [32] for inter-node communication. With coordinated global recovery, nodes can exchange data with each other without having to first perform error correction, since the system can be recovered if an error is detected later.

Checkpoints are coordinated across the system in *logical time* to avoid either quiescing the system or a potentially costly exchange of synchronization messages. Logical time is a time base that respects causality [58]. If Event A causes Event B, then Event A occurs earlier in logical time than Event B. For example, the sending of a message occurs earlier in logical time than the reception of that message. Logical time coordination can be made efficient, since logical time synchronization can be performed without explicit communication beyond what is needed to maintain logical time itself. Moreover, systems can maintain logical time without impacting performance, as will be explained for the logical time bases that we choose. Each node maintains its own logical clock and decides when to take a local checkpoint based on this clock (e.g., every T_c logical cycles). Thus we need a logical time base that allows nodes to independently make the same decisions about which coherence transactions occur in which checkpoint intervals.

First, we solve the easier problem of developing a logical time base that enables consistent viewpoints of when individual coherence *requests* (not coherence *transactions*, which include the request as well as the response and any other messages incurred by the request) occur in logical time. We consider a request to occur when it is processed by the owner, for reasons that will become clear later. Many valid bases of logical time exist. A simple example in a broadcast snooping system is for each component to count the number of coherence requests it has processed and use that as its logical time. If components create checkpoints every T_c logical cycles, it is trivial for all components to agree on the interval in which a coherence request occurred.[2] Directory protocols, however, require a different logical time base. If we could distribute a perfectly synchronous physical clock, we would have a viable logical time base in which logical and physical time are the same. In Section 2.2, we relax this requirement by deriving a logical time base from a loosely synchronized (in physical time) *checkpoint clock*.

Beyond ordering individual coherence requests, the logical time base also must ensure that all components can independently determine the checkpoint interval in which any *coherence transaction* (not just its request) occurs. Snooping cache coherence protocols employ two-hop coherence transactions (requestor to owner to requestor), and directory protocols employ both two-hop (requestor to directory to requestor) and three-hop coherence transactions (requestor to directory to owner to requestor). To determine the checkpoint interval to which each transaction belongs, we exploit the key insight that, in retrospect, a cache coherence transaction appears *logically atomic* once it has completed. Before a transaction has completed, however, no point of atomicity exists. For snooping and directory protocols, a coherence transaction's *point of atomicity* occurs when the owner of the requested block processes the request.

---

2. SMPs do not need to be synchronous, i.e., a request does not need to arrive at every node at the same time. Thus, an SMP with this logical time base could have skew in logical time between nodes [65].

Implementations of *SafetyNet* must ensure that all nodes that participate in a coherence transaction know its point of atomicity. Solving this problem is implementation-specific and the details will be discussed in Section 2.2. Logically, though, the owner must send the checkpoint number of the transaction along with the data in response to the request, so that the requestor knows the transaction's point of atomicity. Moreover, in a three-hop transaction, the requestor must then notify the directory, as well, since it also participated in the transaction and must know its point of atomicity. Figure 2-2 illustrates how a directory protocol with *SafetyNet* determines this point. Note that the requestor does not learn the location of the atomicity point until it receives the response that completes the transaction.

To avoid having to checkpoint transient coherence state, *SafetyNet* exploits logical atomicity and disallows recovery to the middle of a coherence transaction before that transaction has successfully completed (i.e., appears atomic). To ensure that the system never recovers to the "middle" of a transaction, the requestor does not agree to validate a checkpoint (i.e., advance the recovery point) until all of its outstanding transactions issued prior to that checkpoint complete successfully. After completion, the transaction appears atomic, so there is no "middle." Furthermore, by waiting for all outstanding transactions issued prior to that checkpoint to complete before validating the checkpoint, *SafetyNet* avoids checkpointing transient coherence states and in-flight messages.

Since logical atomicity only exists in retrospect, at the time a component creates a checkpoint, the checkpoint only defines what *will be* in that component's checkpoint. The component may not yet have seen the data responses that occur later in physical time but which will appear to have occurred before this checkpoint in logical time. That is, the coherence transaction already occurred in logical time, but it has not yet completed in physical time.

2.1.3 Validating Checkpoints and Deallocating Checkpoint State

Checkpoint validation is the process of determining that a checkpoint is error-free and now can be made the new recovery point. Processors and memories coordinate checkpoint validation so that all components recover to the same checkpoint number on a recovery. For example, checkpoint number 3 (CN3) can be validated only if every component agrees that it could be the recovery point, i.e, all execution prior to CN3 was error-free. For a checkpoint interval to be error-free, every transfer of ownership in that interval must complete successfully, by which we mean that the data was transferred error-free to the receiver. Once every component has independently declared that it has received error-free data in response to all of its requests in the interval before the checkpoint, the recovery point is ready to be advanced. However, the recovery point cannot be advanced until a reduction is performed and all nodes are notified that all other

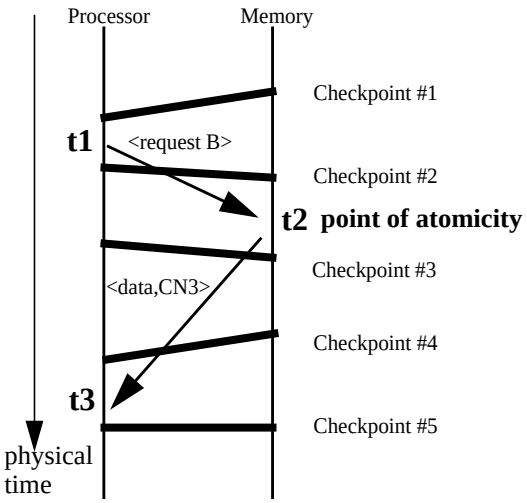

FIGURE 2-2. Example of checkpoint coordination

In this example, physical time flows downwards, and the following assumptions are made:

- Logical time respects causality, so a message cannot be sent in one checkpoint interval and arrive in an earlier interval.
- Checkpoint lines in logical time are not necessarily horizontal, since logical time is not equal to physical time.
- A recovery to checkpoint numbers 2-5 (the duration of the transaction) is not possible until after the transaction, since the processor would not validate any of these checkpoints until the transaction completed successfully at **t3**.
- In practical situations, checkpoint intervals are much longer than typical transaction durations.

At **t1**, the processor issues a request for ownership of block B to the memory, which is currently the owner of the block. The memory processes the request at **t2**, between checkpoints 2 and 3, and defines the transaction's point of atomicity. The directory sends the checkpoint number (CN), CN3, to the requestor, to inform it of the checkpoint to which this transaction belongs. In retrospect, the transaction appears to have occurred atomically at this point. A recovery to checkpoint number (CN) 2 or before would restore ownership to the memory. A recovery to CN 3 or later would maintain ownership at the processor.

nodes are also ready to advance the recovery point. At this point, all transactions prior to this checkpoint have had their points of atomicity determined. After validation, state for the previous recovery point can be deallocated lazily.

A key to *SafetyNet* performance is that validation can be pipelined and performed in the background, off the critical path. Not only can validation be pipelined with the active execution, it can also be pipelined with the validation of other non-active checkpoints. Keeping validation off the critical path requires two features. First, *SafetyNet* must provide more than two available checkpoint contexts—recovery point, active point, and some number of checkpoints pending validation—as illustrated in Figure 1-1. Second, error detection must use dedicated hardware resources (e.g., hardware to check error detecting codes).

Validation latency depends on error detection latency, since a checkpoint cannot be validated until it has been verified error-free. For the example error of a dead switch in the interconnection network, the detection latency must be at least as long as the requestor's timeout latency. Timeout latency can be many traversals of the interconnect, plus some slack built in for contention delays. Adding to validation latency, validation cannot occur until all nodes have coordinated their validations, and this involves an exchange of messages. Since validation latency may be long, it is important for *SafetyNet* efficiency that it be performed in the background and off the critical path.

2.1.4 Recovering the System to a Consistent Global State

If an error is detected, *SafetyNet* restores the globally consistent recovery point checkpoint. The recovery point represents the consistent state of the system at the *logical time* that this checkpoint was taken. Recovery itself requires that the processors restore their register checkpoints and that the caches and memories unroll their logs to recover the system to the consistent state at the pre-error recovery point. All state associated with transactions in progress at the time of recovery is discarded, since this state is, by definition, unvalidated state that occurs logically after the recovery point. The system can thus either reset the network or sink and discard all active coherence messages.

After recovery, the system reconfigures, if necessary, and resumes execution from the recovery point. For the lost switch example, reconfiguration involves routing around the erroneous switch, as is done in many interconnection networks, such as that of the Compaq Alpha 21364 [68]. For transient faults, no reconfiguration is necessary.

SafetyNet's ability to recover to a consistent state relies upon a couple of assumptions. First, most notably, we assume that the recovery point state is protected. Corruption of this state would prevent recovery. We discuss how we protect this state in Section 2.2.1. Second, we assume that the *SafetyNet* mechanisms

themselves are protected. These mechanisms include the communication of messages regarding validation and recovery, and we discuss these error models and how to tolerate them in Section 4.1.

2.1.5 Input/Output Commit Problems

Since real computer systems interact with the outside world, *SafetyNet* must deal with interactions that go beyond its *sphere of recoverability.* A shared memory multiprocessor protected by *SafetyNet* can recover processor state, memory state, and coherence state. However, it cannot recover input/output (I/O) devices, such as disks, displays, printers, and networks (beyond the system's local interconnection network), since these devices are beyond *SafetyNet*'s scope.

The *output commit problem* [32] requires that only validated, error-free data can be communicated outside of the sphere of recovery. For example, the system cannot communicate unvalidated data with the disks if the effects of this communication cannot be undone through recovery. Thus, checkpoint validation determines when the system can interact with the outside world of input/output devices. The standard solution to the output commit problem is to delay all output events until a validated checkpoint. Implementing I/O with InfiniBand (www.infinibandta.org) is a good match for *SafetyNet*, because I/O transactions are set up in memory and then committed with a "doorbell ring." *SafetyNet* would need to delay only the doorbell ring, which should be acceptable to many types of I/O (e.g., to disks and the Internet). In practice, not all I/O devices need to be treated as carefully. For example, the display could be updated with un-validated state if the recovery latency is shorter than could be perceived by the user. Writes to disks (but not disk control registers) might also be performed speculatively, since these actions are idempotent and do not have side effects.

For higher performance I/O, such as cluster communication, the required I/O performance could dictate the validation latency, rather than vice versa. In turn, this would determine the error detection schemes used and perhaps even the fault model. Once again, though, I/O systems that enable *SafetyNet* to extend its sphere of recoverability would help alleviate this constraint. Infiniband or some other protocol that allows *SafetyNet* to encompass the high performance I/O device, such as QPIP [15] would allow us to hide longer error detection latencies without hurting critical performance needs.

The complementary *input commit problem* states that a system that recovers must deal with the input from the outside world that it received between the recovery point and the time at which it recovered. We adopt the standard solution of logging input from the outside world and replaying it after recovery.

2.1.6 Other Classes of Coherence Protocols and Memory Models

Thus far, we have assumed that *SafetyNet* is being applied to a shared memory multiprocessor that implements sequential consistency with a directory-based cache coherence protocol (referred to commonly as just a directory protocol). Other classes of cache coherence protocols and memory consistency models exist, and we now address the issues involved with implementing *SafetyNet* in these different contexts, respectively. These issues are orthogonal to each other, so we can address different protocols independently from different memory consistency models.

Cache Coherence Protocols. Besides traditional directory and broadcast snooping protocols, there are numerous hybrid protocols. These include multicast snooping [11, 101], bandwidth adaptive snooping [66], and the Compaq AlphaServer GS320 [37]. All of these protocols share the same point of atomicity as directories and broadcast snooping: when the owner processes the request. Determining the basis of logical time, though, may differ from protocol to protocol. For example, multicast snooping cannot use the same logical time basis as broadcast snooping, since not every node observes every request, so nodes would have different views of which request occurs at which logical time. In Section 2.2.2, we will discuss implementations of logical time bases for both directory protocols and broadcast snooping protocols.

Hierarchical multiprocessor organizations use coherence protocols that may require different bases of logical time. Systems like Wildfire [45] and Profusion [115] have hierarchical coherence domains. Determining a basis of logical time in such systems is likely to vary by system.

Memory Consistency Models. We have thus far assumed that our system supports sequential consistency [59], and that assumption has determined what architectural state needs to be checkpointed. Different consistency models, however, may enable hardware optimizations that can add to the state that must be checkpointed. Thus, implementing *SafetyNet* in the context of a system with a more relaxed memory consistency model may require additional effort. However, these systems are still assumed to maintain precise interrupts, which facilitates checkpointing of the processor's architected state. For example, *SafetyNet* does not need to checkpoint store queues (for holding stores that have not been committed yet), since these stores are not yet part of the architected state.

Systems that enforce processor consistency (PC), such as SPARC Total Store Order (TSO) [113] or IA-32 [50], enable the use of FIFO store buffers to hold stores that have committed but not yet been made visible to other nodes[3], and these store buffers comprise architectural state. As such, *SafetyNet* would need to checkpoint them, too. In processor consistent systems, two nodes can have store buffer entries for the same block, and the values of the stores can be different. Thus, the checkpoint of consistent state reflects PC's

allowance of different concurrent block values (i.e., values visible within a node but not visible to other nodes).

Implementing *SafetyNet* on top of systems that support more relaxed memory consistency models, such as Alpha [95] and IA-64 [51], may require saving additional state, depending on the system implementation. For example, the Alpha memory model allows for coalescing store buffers, and these store buffers comprise architectural state that would need to be checkpointed by *SafetyNet*.

2.1.7 Integrating *SafetyNet* with Other Levels of Checkpoint/Recovery

Checkpoint/recovery techniques are used at many different levels in a computer system, and *SafetyNet* must interact with them. At the lowest level, a microprocessor can recover from mis-speculation, such as might occur due to a branch misprediction. This level of checkpoint/recovery is invisible to higher levels, including *SafetyNet*. We would not want to have to use *SafetyNet* to recover the state of the entire system just because of a localized branch misprediction. Branch mispredictions are too frequent to incur the penalty of *SafetyNet* recoveries, even if these recoveries are adequately short for handling more infrequent events such as hardware errors.

At the highest level, software-only techniques provide heavyweight availability in the presence of hardware and even software errors. For example, database management systems use software to ensure that vital database state is never lost or corrupted, even if the computer system fails. Logging of transactions (database transactions, not cache coherence transactions) and two-phase protocols for committing database state to disks help to avoid corruption of data. Preservation of data, however, does not provide availability, since the system may still fail and thus be unavailable. Software-only schemes are oblivious to *SafetyNet* (and all other lower-level schemes).

SafetyNet is complementary to other levels of checkpoint/recovery in a computer system. Different system levels require different approaches, and we do not claim that *SafetyNet* is the answer to all checkpoint/recovery needs. We do, however, claim that *SafetyNet* serves an important role at the level of hardware-only, system-wide checkpoint/recovery. *SafetyNet* works in conjunction with these other levels of checkpoint/recovery to provide the appropriate cost/performance tradeoff for each level.

---

3. While a relaxed memory consistency model *allows* for optimizations, implementations do not have to use them. For example, a sequentially consistent system implementation is a valid implementation of more relaxed memory models, such as processor consistency.

The development of *SafetyNet* involved the creation of another level of checkpoint/recovery, called Multi-version Memory (MVM), that could be a useful basis for future work in supporting speculative execution [99]. MVM only permitted local (intra-node) checkpoint/recovery. This local recovery permitted deeper speculation than possible with intra-core recovery (e.g., for branch prediction), since it extended out to the node's memory hierarchy. MVM could be used for speculating on multiprocessor values, such as with false sharing prediction. MVM would be preferable to *SafetyNet*, because local recovery is quicker than global recovery, and speculation would lead to far more recoveries than errors. Moreover, we would not want to recover the entire system due to a localized misprediction. Local checkpoint/recovery has several limitations, though. First, it subjects all inter-node communication to the output commit problem, thus adding error detection latency to the critical path of inter-node communication. Second, an external request for speculative data requires either stalling or recovering. Third, it does not tolerate errors outside of the local node. However, it may be interesting in the future to pursue a multi-leveled approach, wherein speculation is supported by local checkpoint/recovery, and availability is supported by the global *SafetyNet* presented in this thesis.

2.2 Implementing *SafetyNet*

In this section, we will discuss two particular implementations of the *SafetyNet* abstraction. The two implementations differ in the cache coherence protocol that they use, and they help to distinguish what is fundamental to *SafetyNet* implementations and what is protocol-specific. *SN-Snooping* implements *SafetyNet* in a system with a broadcast snooping cache coherence protocol, and *SN-Directory* implements *SafetyNet* in a system with a directory protocol. Both implementations reflect the goal of incurring low overhead in the common case of error-free execution, while not allocating resources towards optimizing the rare case of recovery.

All *SafetyNet* implementations have to address two requirements.

- Point-Of-Atomicity Requirement: All of the participants in a cache coherence transaction—the requestor, the owner, and perhaps other components (e.g., the home directory) depending on the cache coherence protocol—must agree on when a coherence transaction logically occurs, i.e., its point of atomicity.
- Sufficient-Log-Storage Requirement: The system must be able to avoid deadlock due to running out of space to hold logged changes to the system state.

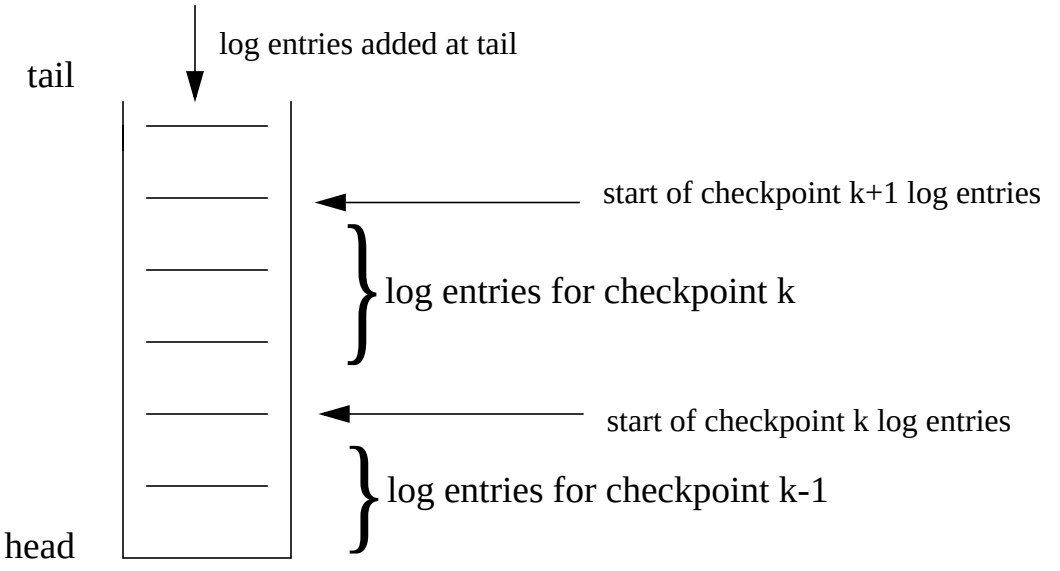

FIGURE 2-3. Checkpoint log buffer (CLB) structure

2.2.1 System Model

SafetyNet systems are composed of some number of nodes connected together by an interconnection network. All nodes contain a CPU, two levels of cache, and a portion of the system's shared memory (which may be separate from the processors). Each CPU has a table of transaction buffer entries (TBEs) for holding state regarding in-progress coherence transactions. A *Checkpoint Log Buffer (CLB)*, associated with each cache hierarchy and memory controller, stores logged state. During error-free execution, the CLB is simply a last-in-first-out (LIFO) write-only buffer. The CLB is illustrated in Figure 2-3. During system recovery, the CLBs are read, but this is the uncommon case.

The system also includes redundant system service processors (which exist in many servers, such as the UltraEnterprise E10000 [18]), which help coordinate advancement of the recovery point as well as system restart after recovery. Recall from Section 2.1.4 that we assume protection of the recovery point state. Recovery point state includes processor registers, caches, CLBs, and memory/directory state. Since corruption of this state would prevent recovery to a error-free consistent state, we protect this state with error correcting codes (ECC).

***SN-Snooping* Specifics.** Figure 2-4 illustrates the *SN-Snooping* system. The nodes are connected in a hierarchical switched interconnection network. Such an interconnect can provide the total order of requests

necessary for snooping, but it does not suffer from the limitations of the shared bus that has been traditionally used in snooping systems.

The snooping protocol is based on a typical MOSI broadcast snooping protocol. The protocol uses two virtual networks: Request and Response. The Request network must be totally ordered, but there are no such constraints on the Response network. There are four types of requests: Get-Shared, Get-Instruction, Get-Exclusive, and Put-Exclusive. Responses are always Data.

***SN-Directory* Specifics.** Figure 2-5 (identical to Figure 1-2) illustrates the *SN-Directory* system. The system's multiple nodes communicate through a two-dimensional torus interconnection network, similar to that used in the Compaq Alpha 21364 interconnection network [68]. One difference worth noting is that switches are composed of two half-switches, one for each direction. Splitting the switches this way offers a redundant path from a node to the rest of the system if one of the half-switches dies, but it results in additional latency to change directions.

The directory protocol is based on a typical MOSI directory protocol. The protocol uses four virtual networks: Request, Forwarded-Request, Response, and Final-Ack. There are four types of requests: Get-Shared, Get-Instruction, Get-Exclusive, and Put-Exclusive. Forwarded requests are either Forwarded-Get-Shared, Forwarded-Get-Instruction, Forwarded-Get-Exclusive, or Put-Exclusive-Ack. Responses are either Data, Ack, or Nack. The Final-Ack network, discussed later, carries Final-Ack and Final-Nack message types. In Appendix A, we specify the protocol in a tabular format developed by Sorin et al [101].

SafetyNet has only a slight impact on the directory cache coherence protocol. Three changes are made so that *SN-Directory* satisfies the two requirements established in Section 2.2, with the first two changes both addressing the Point-Of-Atomicity Requirement. First, when the owner sends a data response message, it labels it with a checkpoint number that defines the transaction's point of atomicity. Second, a Final-Ack network is used in three-hop transactions to notify the directory of the transaction's point of atomicity. When the directory forwards a Get-Exclusive request to the owner, it allocates a transaction buffer entry (TBE).[4] When the requestor receives data (or a nack) from the owner, the requestor sends a Final-Ack (or Final-Nack) to the directory, informing the directory of the point of atomicity and allowing the directory to free its TBE for this transaction. Third, both the directories and processor owners are allowed to negatively acknowledge (nack) requests or forwarded requests, so as to satisfy the Sufficient-Log-Storage Requirement, as will be discussed in Section 2.2.3.

4. If a TBE cannot be allocated, the directory nacks the request.

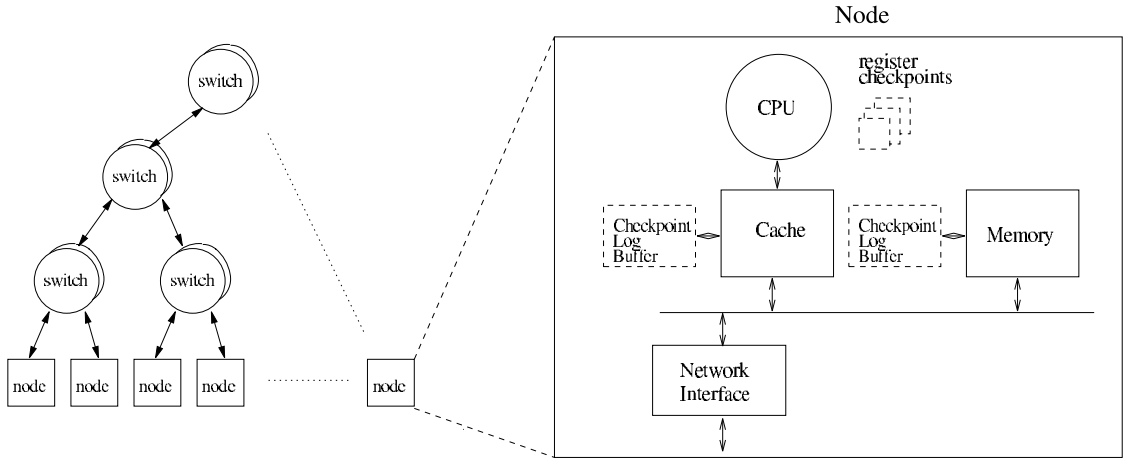

FIGURE 2-4. *SN-Snooping* system model

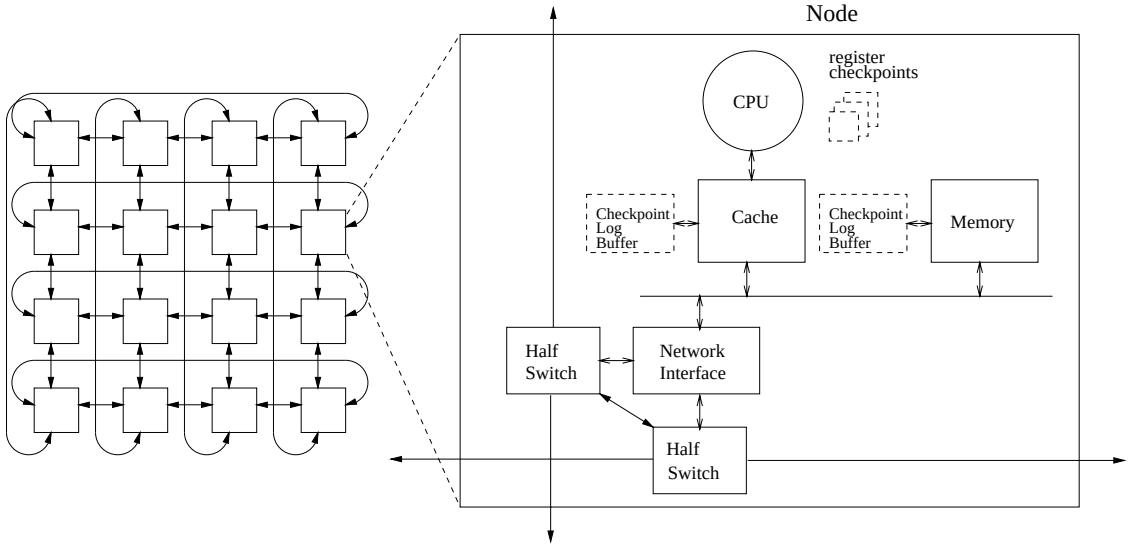

FIGURE 2-5. *SN-Directory* system model

2.2.2 Logical Time Base

As discussed in Section 2.1, we use logical time to address the primary challenge of coordinating checkpoints across a system, which is keeping checkpoints consistent with respect to memory and coherence state. All components must agree, for every coherence transaction, in which checkpoint interval that transaction occurred. Assigning a transaction to a checkpoint interval is protocol-dependent, and it is the most

significant difference in implementing *SafetyNet* on top of different classes of protocols. A similarity, though, is that the point of atomicity in both protocols occurs when the owner of the block processes the request. Note that the ordering point in a directory protocol is different, and it occurs when the directory processes the coherence request (even for a three-hop transaction).

SN-Snooping. A simple logical time base in a broadcast snooping system is for each component to count the number of coherence requests it has processed and use that as its logical time. If components create checkpoints every T_c logical cycles, it is trivial for all components to agree on the interval in which a transaction's request occurred. Moreover, unlike *SN-Directory*, the data response message from the previous owner does not need to include the CN of the point of atomicity, since the requestor already knows it. All nodes can agree that the T_c+1^{th} transaction happened after the checkpoint and the T_c-1^{th} transaction happened before it, so *SN-Snooping* easily satisfies the Point-Of-Atomicity Requirement established in Section 2.2.

SN-Directory. In Section 2.1.2, we discussed how a perfectly synchronous physical clock with zero skew would be a viable basis of logical time for our system with directory coherence. Since that solution is not implementable, we use a loosely synchronous (in physical time) *checkpoint clock* that is distributed redundantly to ensure no single point of failure. On each edge of this clock[5], each component creates a checkpoint and increments its *current checkpoint number (CCN)*. While it might be difficult to distribute a synchronous clock across a system with near-zero skew, it is not nearly so difficult to distribute one with the same frequency and some amount of skew between nodes. As long as the skew between any two nodes is less than the minimum communication time between these nodes, the checkpoint clock is a valid base of logical time, since no message can travel backwards in logical time and thus violate causality. Since communicating messages between nodes entails sending multiple bytes, this time is easily longer than the skew in distributing the edge of a clock. Figure 2-6 illustrates this property. Without this guarantee, the following inconsistency could arise. Consider the case in which processor P1 has a CCN of 3 and sends a request to the owner, P2, while P2's CCN is still 2. Thus, checkpoint 3 would appear to include the reception of the request but not the sending of the request!

To satisfy the Point-Of-Atomicity Requirement, the CN of the point of atomicity must be explicitly exchanged, since nodes cannot infer it as in *SN-Snooping*. Thus, an owner's response to a request includes

5. The frequency of the checkpoint clock is much slower than that of the system clock (e.g., 10 kHz), which simplifies its implementation. We will address this issue in more detail in Section 2.2.4.

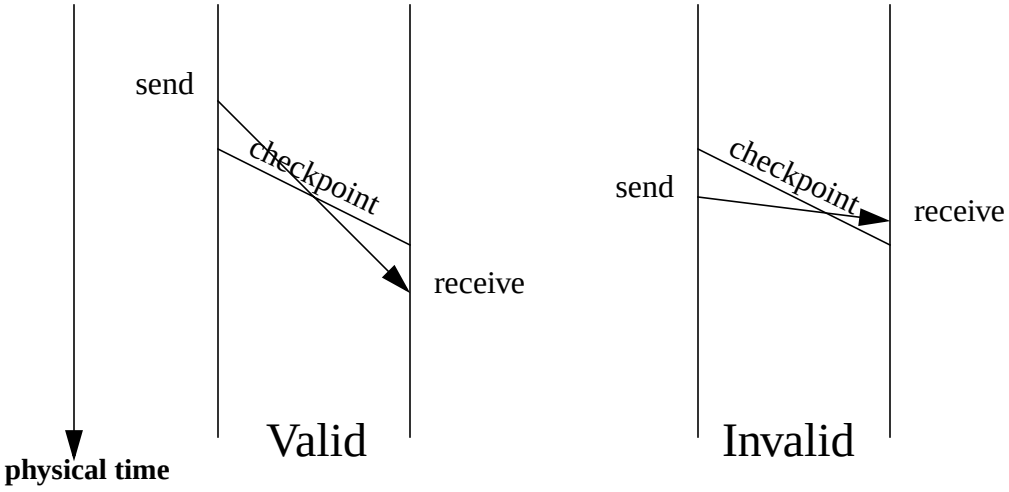

FIGURE 2-6. Ensuring that logical time respects causality

the CN of the point of atomicity and, in a three-hop transaction, the requestor sends a Final-Ack to the directory to inform it of the point of atomicity.

2.2.3 Logging

SafetyNet uses *Checkpoint Log Buffers (CLBs)* to hold incrementally checkpointed memory state. Logically, *SafetyNet* writes a memory block to a CLB whenever an *update-action* (i.e., store or transfer of ownership) might have to be undone in the case of a recovery. Since caches perform stores and both caches and memories can transfer ownership of blocks, each of these components has a CLB. Except during recovery, the CLBs are write-only and off the critical path.

To reduce storage and bandwidth requirements, *SafetyNet* caches (but not memories) only log a block on the first update-action per checkpoint interval. To detect this case, *SafetyNet* adds a *checkpoint number (CN)* to each block in the cache, denoting to which checkpoint it belongs. Initially, all CNs are set to null. On each update-action, *SafetyNet* (1) compares the component's current checkpoint number (CCN) with the block's CN, (2) logs the block, if necessary, (3) updates the block's CN to CCN+1, and (4) performs the update-action. Blocks must be logged to the CLB if the block's CN=null or CCN ≥ CN. For example, a store by a processor with CCN=3 to a block with CN=4 need not be logged. Blocks with null CNs have not been written or transferred recently, and they implicitly belong to the recovery point as well as all subsequent checkpoints. Having CNs on blocks enables logic to determine whether logging of a store or transferring ownership to another node would be redundant.[6] Figure 2-7 illustrates an example of logging at a

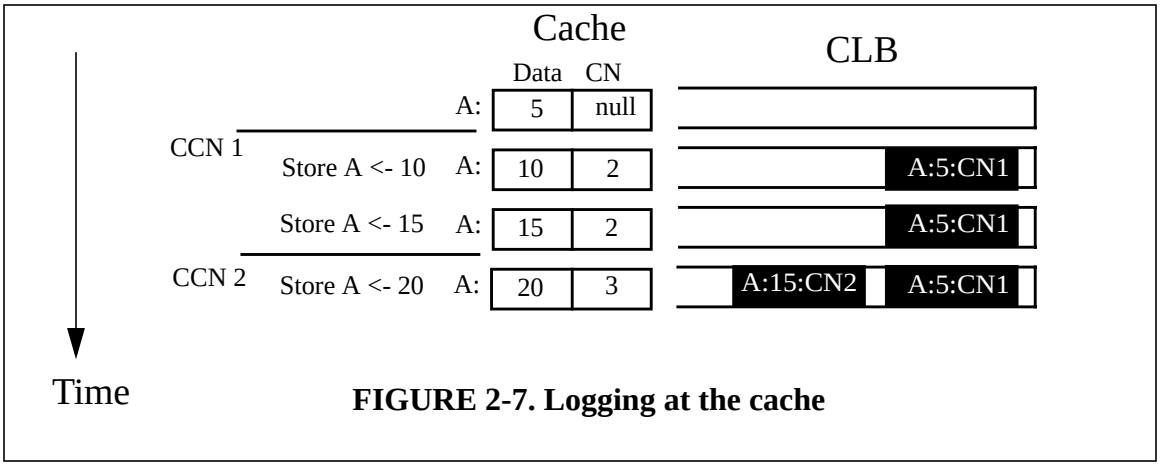

FIGURE 2-7. Logging at the cache

cache. In Section 2.2.7, we discuss efficient ways to store and manipulate CNs at the caches. CNs are not needed on CLB blocks, but they are shown in Figure 2-7 for illustrative purposes. A CLB implementation only requires a head and tail pointer for each checkpoint number, as was shown in Figure 2-3.

When giving up ownership of a block, a component performs logging (as described above) and then sends the block *with the updated CN* to the requestor. This policy follows from a key insight from Wu et al. [120]: a transfer of ownership is just like a write, in that we cannot be sure that it will not be undone by a recovery. Consider the case in which P1 transfers ownership of block B to P2 with B's CN set to 3 (i.e., P1's CCN is 2) and P2 wishes to then perform a store to it while its CCN is still 2. Logging is unnecessary, since P2 was not the owner at checkpoint 2. This is the same as if P1 owned block B with CN=3 and performed a store to it while its CCN is still 2.

The CLBs can be sized for performance and not correctness, since *SafetyNet* can avoid situations in which the CLB fills up and violates the Sufficient-Log-Storage Requirement established in Section 2.2. Even when it appears that an entry must be logged in the CLB, logging can be avoided. In the case of store overwrites, we can throttle requests from the CPU, thus stalling the CPU. In the case of coherence ownership transfers, there are two situations. First, there are acquisitions of ownership, and these can be throttled by the requestor. The tougher situation is relinquishing ownership, since the owner has no control over this. For *SN-Snooping*, where checkpoint intervals are known *a priori* to be T_c transactions long, the solution is to throttle stores when CLB space reaches the minimum necessary for all possible transactions that could still occur in the current interval. For *SN-Directory*, where a component does not know how many more transactions could arrive in an interval, the solution is for the owner to negatively acknowledge (nack)

6. There are other optimizations for reducing logging due to *attaining* ownership, but they are less important. The key is reducing logging due to store overwrites.

coherence requests. Nacking avoids the transfer of coherence ownership and its associated log entry. In *SN-Directory*, either the directory or another cache can be the owner that needs to nack a request. While directory nacks are common in many directory protocols, nacks from caches are rarer. In this case, the directory has forwarded the request to the owner cache, and the owner cache sends a nack to the requestor. The requestor then must send a Final-Nack to the directory to inform it that the transaction has been nacked. This Final-Nack uses the same virtual network used to carry Final-Ack messages. Upon reception of a Final-Nack, the directory then reverts its state back to the state from before the nacked transaction and deallocates its TBE.

To the first order, CLB occupancy is unrelated to cache size or memory size. CLB occupancy is a function of the workload intensity. More specifically, it is a function of the number of update-actions to distinct blocks per checkpoint interval. It might appear that a smaller cache would cause more update-actions because of additional cache replacements. However, even if a block is replaced to memory and retrieved from memory multiple times in an interval, each of which is an update-action, only the first such update-action per interval is logged. It also might appear that a larger memory would place more pressure on the CLBs. However, the parameter that matters is the amount of memory touched in an interval.

Having to stall stores or acquisitions of ownership due to a full CLB degrades performance, but it does not affect correctness or lead to deadlock or livelock. Livelock would occur if execution was stalled—stores were stalled and all cache coherence requests were nacked—and execution could only be un-stalled if forward progress was made. The key to avoiding livelock is that CLB space will free up *independently* of execution. CLB space is freed when an old, pending checkpoint is validated. When this checkpoint becomes the new recovery point, the old recovery point state (including CLB entries) can be discarded, as will be discussed in Section 2.2.5. Thus, the freeing of CLB space is tied to checkpoint validation (i.e., error detection) and not active execution. The performance impact of stalls due to full CLBs will be evaluated in Section 3.4.1.

2.2.4 Checkpoint Creation

Checkpoint creation is kept lightweight, since it is a common-case event that occurs on each edge of the checkpoint clock. A processor checkpoints its non-memory architectural state (i.e., registers) and increments its CCN. A memory controller simply increments its CCN. Checkpointing of memory and coherence state is achieved through logging, so no checkpointing of that state is necessary at checkpoint creation.

Since checkpoint numbers are encoded in a finite number of bits, say k, we can only have 2^k checkpoint contexts. In all experiments shown later, k is two (i.e., we can only have four checkpoint contexts).[7] CN wraparound can only occur if validation ceases (i.e., because a coherence transaction does not complete) while checkpoint creation continues. We avoid wraparound by choosing a request timeout latency that is shorter than the latency to wraparound. Thus, a request would timeout, and thus trigger a system recovery, before it could stall validation to the point at which wraparound could occur.

Checkpoint creation policy is simply choosing a suitable checkpoint period, T_c. For *SN-Snooping*, the checkpoint period is the number of coherence transactions per checkpoint interval. To keep the period bounded in physical time and limit output commit latency, the service processor periodically injects null coherence requests if it observes that no requests are being made. For *SN-Directory*, the checkpoint period is the reciprocal of the checkpoint clock frequency, f_c. As f_c decreases (given a constant number of outstanding checkpoints), *SafetyNet* can tolerate longer error detection latencies. For example, we allow four outstanding checkpoints and choose f_c equal to 10 kHz (i.e., T_c is 100,000 processor cycles at a processor clock of 1 GHz) to enable 400,000 cycles (0.4 msec) of detection latency tolerance.

The cost of increasing tolerable detection latency is more pressure on the CLBs and longer delays due to the output commit problem. While increasing T_c allows for more compression of logged data, since only the first of multiple writes or ownership transfers in a checkpoint interval requires logging, total CLB storage is a function both of logging frequency and interval length. However, given sufficient CLB storage, the value of T_c has little effect on common-case performance, as will be shown in Chapter 3. The choice of T_c has a greater impact on I/O performance, as was discussed in Section 2.1.5.

2.2.5 Checkpoint Validation and Deallocation of Checkpoint State

Checkpoint validation requires that all components agree that execution up until that checkpoint was error-free. For a given error model and associated error detection mechanisms, each component waits to ensure that the error detection mechanisms report no errors for activity prior to the checkpoint to be validated. We now address the error model of a lost/corrupted message that is detected with a timeout at the requestor, since this is likely to be the longest latency detection mechanism. Other long latency mechanisms may be implemented, including strong error detecting codes applied to incoming messages, but these should presumably be shorter than timeouts.

---

7. As will be discussed in Section 2.2.7, we use unary encoding of checkpoint numbers at the caches. Thus, we need four bits to encode the four possible checkpoint numbers.

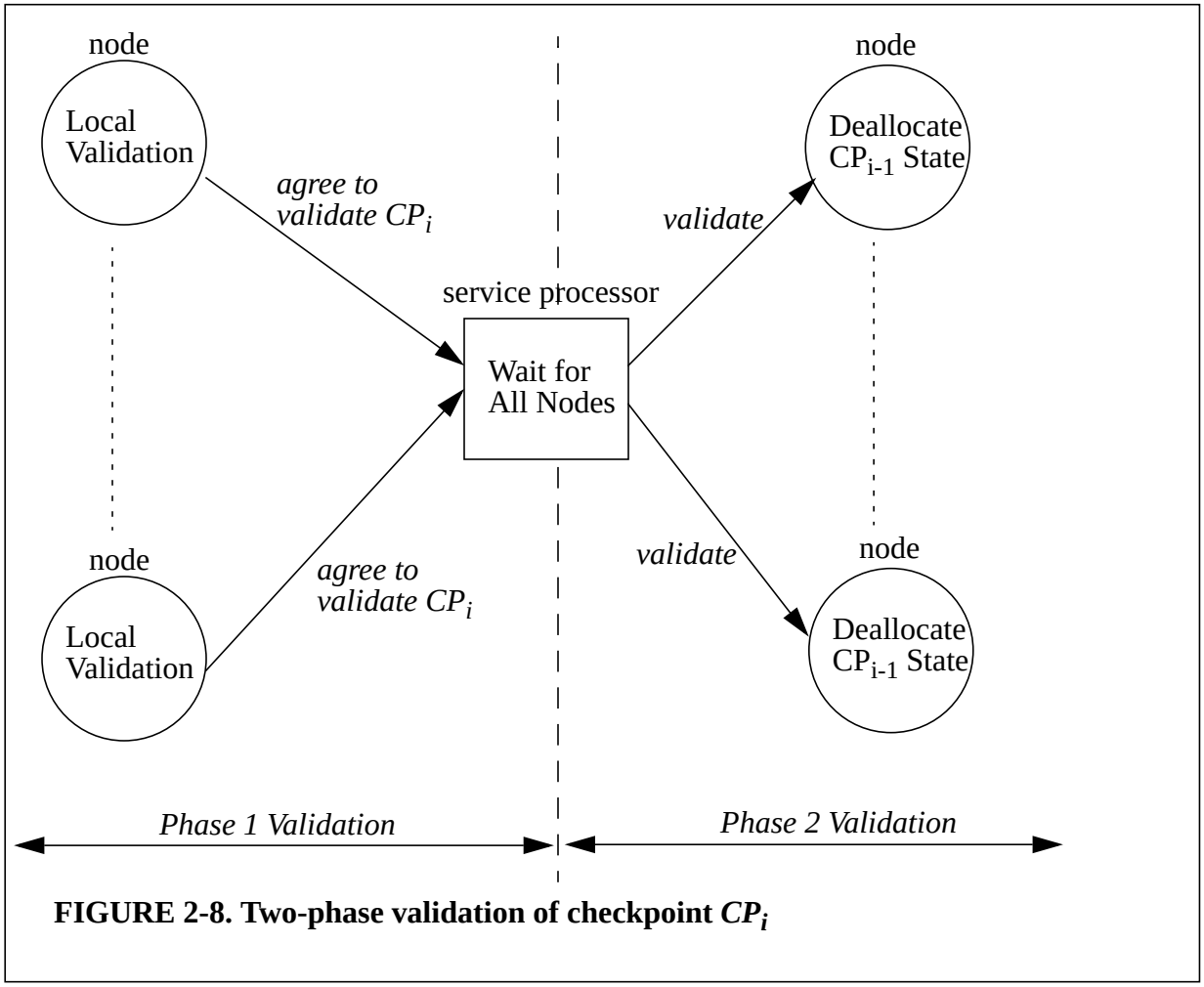

FIGURE 2-8. Two-phase validation of checkpoint CP_i

A cache controller only agrees to validate a checkpoint once every transaction it initiated in the interval before that checkpoint completed successfully. Thus, if a cache controller checks its TBE table and determines that none of its outstanding requests were issued in the checkpoint to be validated, then it can validate that checkpoint.

A directory controller only agrees to validate a checkpoint once every transaction for which it forwarded a request to a processor owner (i.e., three-hop transaction) completed successfully. Thus, the requestor must send a Final-Ack (or Final-Nack) to the directory after its request has been satisfied (or nacked), so that the directory can deallocate its TBE for the transaction. Any lost message will prevent advancement of the recovery point. If the recovery point cannot be advanced after a given amount of time, the system assumes an error has occurred (such as a lost message) and triggers a system recovery.

We coordinate global validation with a two-phase scheme illustrated in Figure 2-8. A component informs the service processor that it is ready to advance the recovery point. Once every component has informed the service processor that it is ready to advance the recovery point, the service processor broadcasts the

new *recovery point checkpoint number (RPCN)*.[8] Execution does not slow down while checkpoints are validated in the background, similar to a fuzzy barrier [43].

Processor and memory controllers deallocate a checkpoint, say CP_i, by discarding their now unneeded architectural checkpoints for checkpoint CP_{i-1}. We discard the state for the before images that enable the system to recover from CP_i to CP_{i-1}, since we will never need to recover to CP_{i-1} now that CP_i is the recovery point. A processor discards its register checkpoint for CP_{i-1}. Caches deallocate a checkpoint by clearing the CN of all blocks that had CN set to CP_{i-1}. We discuss how to implement this feature in Section 2.2.7. Caches and memories discard logged data at their CLBs from CP_{i-1}.

2.2.6 System Recovery and Restart

If a component detects an error, it notifies the service processor to trigger a recovery. The recovery message, which includes the RPCN, is broadcast (redundantly) by the service processor, and all nodes proceed to recover to the recovery point. The process of recovery involves several steps, and it leverages the insight that the state of any transactions in progress, by definition, is unvalidated state that is now discarded. First, the interconnection network is drained, and all state related to coherence transactions that were in progress at the time of the recovery, including TBE state for blocks the node is trying to acquire, is discarded. Second, processors, caches, and memories recover the RPCN checkpoints. Memories just sequentially undo the logged update-actions in their CLBs. Undoing the memory CLB involves copying the blocks from the CLB to the memory, traversing the CLB from the tail to the head (i.e., in reverse order of insertion). Processors restore their register checkpoints. Caches invalidate all blocks written or sent in an unvalidated checkpoint interval (i.e., blocks with non-null CNs), and they undo the logged actions in their CLBs. Similar to memory, undoing the cache CLB involves copying the blocks in reverse order from the CLB to the cache.

At recovery time, the TBEs may also hold validated state that cannot be discarded because it is part of the recovery point. During normal execution, on a cache replacement of an owned block, the block is moved from the cache to the TBE while the Put-Exclusive is pending, so that the cache frame can be re-used immediately. Thus, the only copy of the block is in the TBE. If this block is validated state, it is part of the recovery point, and it must be retained during the system recovery process. During a recovery, we handle this block by re-issuing a Put-Exclusive for it and leaving it in the TBE. Trying instead to push the block back into the cache is legal but more difficult.

---

8. Communication of coordination messages (which are infrequent) can be made reliable through redundancy, if this double fault model is to be tolerated. We will discuss this issue in Section 4.1.5.

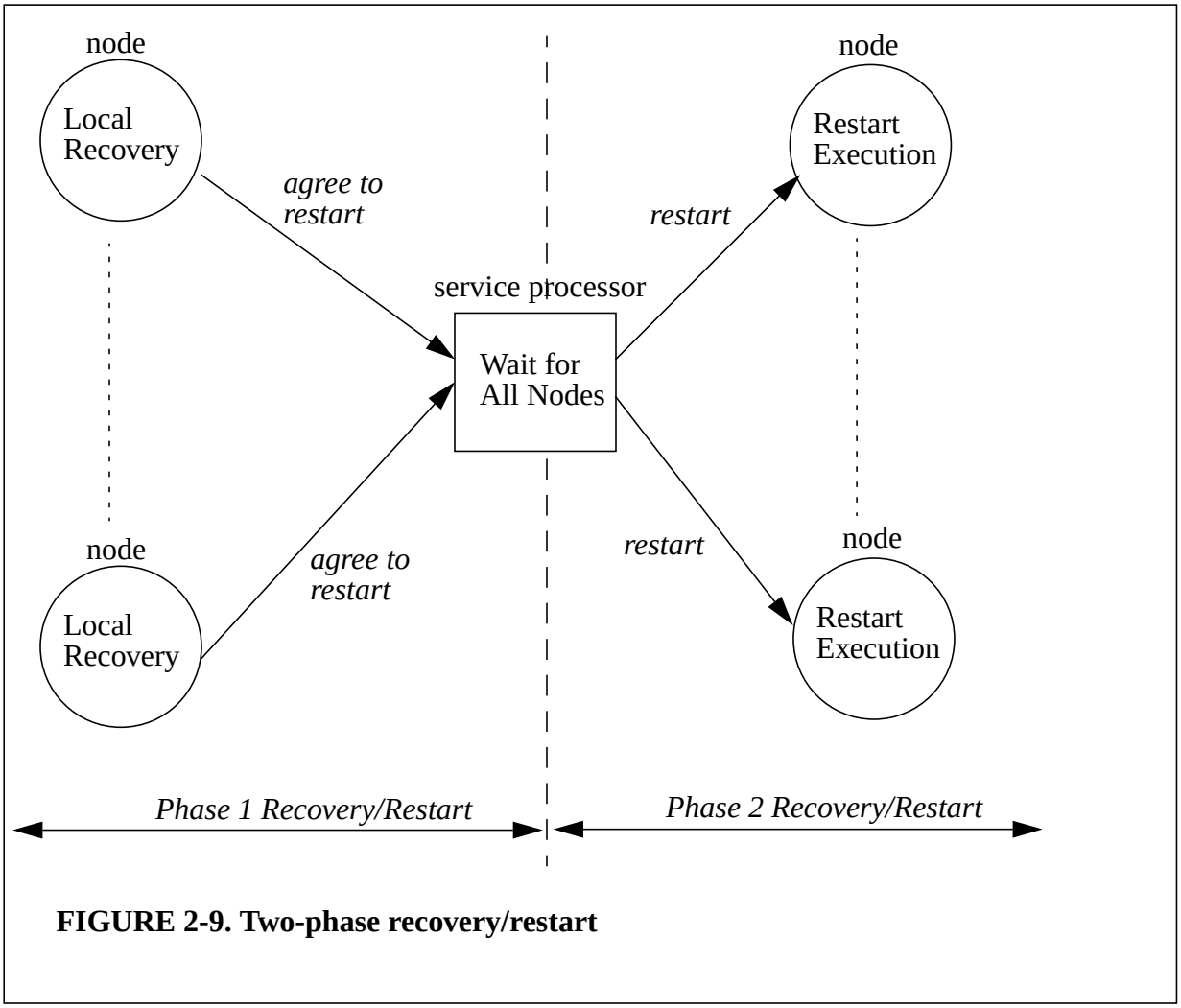

FIGURE 2-9. Two-phase recovery/restart

After recovery and reconfiguration (if needed), a restart message is broadcast to inform the nodes that they can resume operation. The restart cannot begin until every node has finished its recovery. As with coordination to validate checkpoints, we implement a two-phase coordination in which every node informs the system service processor once it is ready to restart and then the service processor broadcasts the restart message. This two-phase restart is illustrated in Figure 2-9.

2.2.7 Implementation Details

In this section, we discuss some of the implementation details that have been omitted thus far in Section 2.2, including how to maintain checkpoint numbers at the cache and how to checkpoint the processor register state.

TABLE 2-1. Modifications to *SafetyNet* cache behavior

Operation	Action
Load	*nothing*
Store	If CN=null or CCN≥CN, then log old copy of block in CLB
Coherence Transfer	If CN=null or CCN≥CN, then log old copy of block in CLB
Checkpoint Validation	Clear CN bit for all blocks in cache
System Recovery	Invalidate all blocks with non-null CNs

Checkpoint Numbers at Cache. We now describe how to store and manipulate checkpoint numbers at the caches. Caches maintain checkpoint numbers to enable optimized logging of update-events. It is crucial to optimize logging at the L1 cache, since stores are so frequent, but optimizing update-events at lower levels of the cache hierarchy is less crucial. Thus, the following cache modifications could be eliminated at caches below the L1, if the cost of implementation is deemed to be not worth the benefit.

Cache operation is conventional, with three important exceptions: (1) a store hit may trigger logging of the block that would be overwritten, (2) a validation of checkpoint *i* must find blocks with CN=*i* and then set CN=*null*, and (3) a recovery must invalidate blocks with CN≠*null*. Since we always recover to RPCN, there are no partial rollbacks. Case (1) can be detected by comparing the processor's CCN and the stored block's CN in parallel with a standard tag comparison. A store to the cache thus reads the cache tags (but not the data) before writing it, but this is also true for normal stores, since they require a tag lookup.

Checkpoint validations and system recoveries can be made to operate globally on the caches in constant time with two changes. First, we store checkpoint numbers encoded as one-hot bit vectors. This encoding requires *k* bits to support *k* active checkpoints, which is not a problem for the small *k* we envision (e.g., four). Second, we keep the checkpoint numbers in the same SRAM with the cache tags and augment the cache tags with a flash clear on each CN bit column, similar to the mechanism in caches with flash invalidation [61].

We summarize *SafetyNet* modifications to cache behavior in Table 2-1.

Register Checkpointing. We now discuss specific details of how to checkpoint the processor register state. There are many possibilities for doing this, and the best design choice depends on the performance required and the cost of implementation. In Section 3.4.3, we will demonstrate that *SafetyNet* performance is quite insensitive to register checkpointing latency for the checkpoint interval of 100,000 cycles that we consider, since checkpointing occurs so infrequently. Thus, we choose a simple, unoptimized design that

leverages existing datapaths. A processor maintains register checkpoint contexts that are accessible by the load/store functional unit. On a checkpoint, the processor uses the load/store unit to sequentially copy the architected registers into a register checkpoint context. On a recovery, the load/store unit extracts the recovery point register checkpoint and copies it into the architected registers. Unlike more traditional shadow registers, these do not require extra datapaths between the architected registers and the shadow registers. Nor do these shadow registers require low-latency accessibility.

For shorter checkpoint intervals, a more optimized register checkpointing scheme may be necessary to avoid performance degradation. We could employ more traditional (i.e., fast) shadow registers, at the cost of adding this datapath and using valuable space near the register file, but we do not explore this issue in this thesis.

2.2.8 Summary of Implementation

We have developed two particular implementations of the *SafetyNet* abstraction. The implementations address the three challenges that were raised for logging schemes. First, we exploit checkpoint granularity to reduce the amount of logging necessary. Second, we efficiently coordinate checkpoints across the system in logical time, with a different logical time base for each implementation. Third, we enable checkpoint validation to be performed in the background, thus hiding the potentially lengthy latency of error detection (e.g., for timeouts on coherence requests).

These implementations require three changes to the processor and caches. First, the processor must be able to checkpoint its register state. While modifying the core may be undesirable, register checkpointing is not performance-critical, since it is uncommon, and copying out registers is straightforward if it does not need to be fast (we will assume 100 cycles in later results). Second, we must be able to copy old versions of blocks out of the cache before overwriting or transferring them. This increases cache bandwidth, but we will show in Chapter 3 that the increase is a small fraction of cache bandwidth used for our commercial workloads. Third, we add CNs to cache blocks (at least at the L1 cache), to enable optimized logging. Adding CNs to cache blocks requires customization of the cache design in order to support flash clear.

SN-Directory also requires three changes to the underlying directory coherence protocol. First, we add checkpoint numbers on data response messages, so that the requestor knows the transaction’s point of atomicity. Second, we allow both directories and processors to nack coherence requests, in order to avoid filling a CLB. Third, we add a Final-Ack (or Final-Nack) from the requestor to the directory on three-hop coherence transactions, so that the directory knows in which checkpoint interval the transaction occurred (or was nacked).

Other implementations of *SafetyNet* are certainly possible, but *SN-Snooping* and *SN-Directory* demonstrate that *SafetyNet* can be applied to the two primary classes of cache coherence protocols.

2.3 *SafetyNet* Conclusions

In this chapter, we developed a scheme, called *SafetyNet*, that enables globally consistent checkpoint/recovery. We also describe two specific implementations of *SafetyNet*. In developing *SafetyNet*, we make three contributions which provide the intuition for why Chapter 3 will show *SafetyNet* to be efficient in the common case of error-free execution.

- *SafetyNet* efficiently coordinates the creation of checkpoints across the system in logical time.
- *SafetyNet* minimizes the amount of state that must be checkpointed through the use of optimized logging.
- *SafetyNet* hides the latency of error detection by pipelining the validation of checkpoints in the background. The system can continue to execute while it determines if old checkpoints can be validated.

Chapter 3

SafetyNet Evaluation

In this chapter, we evaluate *SafetyNet*. Our primary focus is on the performance impact of implementing *SafetyNet*, since *SafetyNet* is more likely to be widely used if it does not significantly degrade error-free performance. Performance is a function of many system and workload parameters, and we explore how these parameters affect performance. The exploration of the system parameter space sheds light not only on performance but also on hardware cost. For example, CLB sizing is important to achieving performance and it also contributes directly to cost. If the only way to achieve acceptable performance was to use unreasonably large CLBs, which is fortunately not the case, then that cost would decrease the viability of deploying *SafetyNet*. There are other costs of implementing *SafetyNet*, such as extra cache bandwidth for logging, and we explore the system parameters that constitute the primary costs.

In Section 3.1, we develop a qualitative model of system performance. The model serves to illustrate the system and workload parameters and how they contribute to *SafetyNet* performance. While the model is not intended as a tool for quantitatively predicting performance, we will refer back to the model during the quantitative analysis to provide insight into the results.

In Section 3.2, we describe our methodology for quantitatively evaluating *SafetyNet*. Since this thesis addresses commercial servers, we use full-system simulation in order to run commercial workloads, including database and web server workloads. While we focus on commercial workloads in this thesis, *SafetyNet* can also be applied to other workloads. We evaluate one scientific benchmark, for comparison, and we discuss issues involved in supporting other types of workloads.

In Section 3.3, we determine *SafetyNet* performance by running three experiments in which we compare the performance of *SafetyNet* versus that of a system unprotected from faults. The two key results are that:

- Differences in performance between *SafetyNet* and an unprotected system are statistically insignificant.
- *SafetyNet* continues to run in the presence of hard and soft faults.

To ensure that the design is not overly sensitive to specific implementation parameters, we present sensitivity analysis in Section 3.4. This analysis will show that *SafetyNet* performance is somewhat dependent on

CLB sizing and the checkpoint interval, but it is not sensitive to the latency for register checkpointing. Analysis will also demonstrate that the additional cache bandwidth required by *SafetyNet* is negligible. Lastly, analysis will illustrate how *SafetyNet* performs as a function of the soft error rate.

3.1 High-Level Performance Model

In this section, we present a high-level model of *SafetyNet* performance. The purpose of the model is to reveal the system and workload parameters that play important roles in determining system performance and to illustrate qualitatively how they interact. In the quantitative performance analyses in Section 3.3 and Section 3.4, we will refer back to this model to provide insight into the results. This model is not intended to serve as a tool for evaluating detailed design decisions.

The equations for performance are functions of system and workload parameters. The key parameters are the following:

- checkpoint period (T_C)
- CLB size *(CLBSize)*
- workload intensity, i.e., frequencies of stores and coherence requests *(WorkloadIntensity)*
- processor register checkpointing latency *(CheckpointLatency)*
- error rate *(ErrorRate)*

Much of the sensitivity analysis in Section 3.4 will involve varying one of these parameters while holding the others constant. Other significant parameters exist, but we will consider them to be fixed in this evaluation. These parameters include the number of checkpoint contexts (fixed at 4), processor speed, cache sizes, and interconnection network bandwidth. All of these parameters and their values will be listed in Section 3.2.1.

3.1.1 Error-Free Performance

Error-free *SafetyNet* runtime would be equivalent to the error-free runtime of an unprotected system, except for the three issues illustrated in the following equation:

$$Runtime(SN) = Runtime(Unprotected) + CLBStalls + CPOverhead + MiscOverhead \quad \textbf{(EQ 1)}$$

The first issue is stall time due to filling a CLB. CLB stall time, in turn, is directly related to the interval length (T_C) and workload intensity, and it is inversely related to the CLB size. The following function for

CLB stall time is non-linear (f_{NL}), but we place variables in the numerator and denominator to reflect whether the function is directly proportional or inversely proportional to the variables.

$$CLBStallTime = f_{NL}\left(\frac{T_C \times WorkloadIntensity}{CLBSize}\right) \quad \textbf{(EQ 2)}$$

The non-linearity in this function has a couple of causes. First, as T_c or workload intensity increases, there are more update-events in the interval, although this is not a linear relationship due to optimized logging. Workload intensity is also not stationary, and bursts in intensity can cause bursts of CLB stalls, even though the CLB size may be sufficient for the mean workload intensity. However, since CLB storage is dynamically allocated to checkpoints (i.e., each of *N* checkpoint contexts is *not* statically assigned *1/Nth* of the CLB), a burst of high frequency logging could be counterbalanced by a stretch of lower frequency logging. Second, as CLB size increases, there are fewer CLB stalls, yet the relationship is non-linear. There is an inflection point in the curve of CLB stalls as a function of CLB size, at the point at which the CLB is large enough to not fill often. Once the CLB size is sufficiently large, increasing it further offers no benefit. However, if the CLB size is particularly small, the system is highly susceptible to bursts of CLB stalls.

The second issue is overhead due to checkpointing. This overhead is a linear function (f_L) of the processor register checkpointing latency divided by T_c.

$$CheckpointOverhead = f_L\left(\frac{CheckpointLatency}{T_C}\right) \quad \textbf{(EQ 3)}$$

The third issue incorporates all other *SafetyNet* overheads. These overheads include many negligible latencies (e.g., extra latency due to extra congestion from checkpoint coordination messages), but the primary potential component is extra latency for stores that require logging in the CLBs. The overhead is directly related to the workload intensity, since an increased rate of stores leads to an increased rate of stores that require logging, although the latter increase is less than the former due to optimized logging of stores. Variance in the workload intensity is less important The overhead is inversely proportional to the checkpoint interval, since a longer interval contributes to more optimized logging of stores. The overhead is a non-linear function of these two parameters.

$$MiscOverhead = f_{NL}\left(\frac{WorkloadIntensity}{Tc}\right) \quad \textbf{(EQ 4)}$$

The non-linearity can be explained with two extreme examples. First, the workload intensity can increase without any corresponding increase in the rate of stores that require logging, if the logging optimization hides the additional stores. Second, T_c can increase without any corresponding decrease in the rate of stores that require logging, if the longer checkpoint interval does not contribute at all to the logging optimization. While neither of these extreme examples is likely, they still illustrate the potential non-linearity in this overhead function.

3.1.2 Performance in Presence of Errors

When errors occur, the unprotected system fails and *SafetyNet* continues to perform at a degraded level. The impact upon *SafetyNet* performance due to errors is a function of the error rate and the recovery time. The effects are captured in the following equation:

$$Runtime(WithErrors) = Runtime(NoErrors) + ErrorRate \times RecoveryTime \quad \textbf{(EQ 5)}$$

Recovery latency consists of four components: discarding unvalidated checkpoint state, restoring the state from the recovery point, re-configuring (e.g., changing the routing to avoid a dead switch) if necessary, and re-executing the work that was lost between the recovery point and the fault.

$$RecoveryTime = Discard + Restore + Reconfigure + ReplayWork \quad \textbf{(EQ 6)}$$

We do not evaluate reconfiguration latency since it is both difficult to estimate and it is only a one-time penalty, although it could be a large penalty. The latencies to restore the recovery point and to replay lost work are both functions of the checkpoint interval and the workload intensity. Re-executing lost work is likely to dominate, since the recovery point can be hundreds of thousands of cycles in the past. *SafetyNet* can tolerate longer error detection latencies with less frequent (i.e., larger) checkpoints, but it does so at the cost of more potential lost work.[1] Nevertheless, even a one million cycle recovery latency is still only one millisecond (i.e., much shorter than a failure and reboot).

3.2 Methodology

In this section, we describe both our full-system simulation infrastructure and the commercial workloads with which we evaluate *SafetyNet*. Since this research addresses commercial servers, we must evaluate our

1. The most significant costs of larger checkpoint intervals are the additional CLB size requirements and longer output commit penalty.

work using important commercial workloads. To simulate systems that can run these workloads requires full-system simulation.

3.2.1 Simulation Infrastructure and Target System

We simulate a 16-processor target system with the Simics full-system, multiprocessor, functional simulator [64], and we extend Simics with a memory hierarchy simulator to compute execution times.

Simics. Simics is a system-level architectural simulator developed by Virtutech AB that can boot unmodified commercial operating systems and run arbitrary unmodified applications. We use Simics/sun4u, which can simulate Sun Microsystems's SPARC V9 platform architecture (e.g., used for Sun E6000s) in sufficient detail to boot an unmodified copy of Sun Solaris 8. Simics is a functional simulator only, and it assumes that each instruction takes one simulated cycle to execute (although I/O may take longer), but it provides an interface to support detailed memory hierarchy simulation.

Processor Model. We use Simics to model a processor core that, given a perfect memory system, would execute four billion instructions per second and generate blocking requests to the L1 cache and beyond. We use this simple processor model to enable tractable simulation times for full-system simulation of commercial workloads. While an out-of-order processor model might have an impact on the absolute values of the results, it would not qualitatively change them (e.g., whether a failure is avoided). For evaluating processor/cache overhead for checkpointing register state, we model a conservative latency of 100 cycles.[2] We

TABLE 3-1. Target system parameters

L1 Caches (I and D)	128 KB, 4-way set associative
L2 Cache	4 MB, 4-way set associative, 4ns
Memory	2 GB, 64 byte blocks, 80ns
Miss From Memory	180 ns (minimum, uncontended, 2-hop)
Checkpoint Log Buffer	512 kbytes total, 72 byte entries
Interconnection Network	2D torus, link bandwidth = 6.4 GB/sec
Checkpoint Interval	100,000 cycles = 100 μsec

2. If checkpointing was a more frequent event (e.g., if we were using *SafetyNet* to support speculation), we could optimize register checkpointing latency by using shadow register copies. As explained in Section 2.2.7 and evaluated in Section 3.4.3, *SafetyNet* performance is insensitive to this latency, so we do not need to optimize it.

conservatively charge eight cycles for logging store overwrites (8 bytes/cycle for 64 byte cache blocks), but store overwrites that require logging comprise only about 0.1% of instructions.

Memory Model. While we have developed two implementations of *SafetyNet*, we only evaluate *SN-Directory* in this section. We have evaluated *SN-Snooping*, but the results are similar enough not to warrant their presentation here. We have implemented a memory hierarchy simulator that supports the *SN-Directory* protocol as well as a comparable protocol without *SafetyNet* support. The simulator captures all state transitions (including transient states) of our coherence protocols in the cache and memory controllers. We simulate a two-level cache hierarchy with a split L1 cache. We enforce exclusion between the L1 caches and the L2 cache. In Table 3-1, we present the design parameters of our target memory systems. With a checkpoint interval of 100,000 cycles and four checkpoint contexts, *SafetyNet* can tolerate error detection latencies up to 400,000 cycles (0.4 msec at 1GHz).

We model a two-dimensional torus interconnection network, and we model link latency and contention within this interconnect, including contention due to checkpoint validation coordination messages. The interconnection network's routing is statically determined. The switches within the network are composed of two half-switches, one for the north/south direction and one for the east/west direction. This design provides sufficient redundancy in the case that a half-switch is lost (as will be explored in greater depth in Section 3.3.3), but it incurs extra latency to change directions in a switch.

The memory system, including the coherence protocol, was specified in a domain specific language called SLICC (Specification Language for Implementing Cache Coherence) that was developed by Milo Martin at the University of Wisconsin. Protocols were developed using the tabular specification methodology proposed by Sorin et al [101]. To further exercise the protocol implementations, we drove them for billions of cycles with a random tester that injected errors and stressed corner cases by exploiting false sharing and reordering messages [119]. We did not perform more formal verification of *SafetyNet* protocols, but we have performed manual formal verification of other related protocols [22, 77, 101], and this verification work has influenced the design of the *SafetyNet* protocols.

I/O Model. As will be discussed in Section 3.2.2, our workloads are scaled down in size to run on our simulator. In particular, our workloads are scaled to run mostly in memory. The I/O that does still occur is not handled by our memory system simulator. Instead, it is handled strictly by Simics, which provides functionality but little timing fidelity.

Recovery. Since Simics cannot currently be recovered to a point hundreds of thousands of cycles in the past, we must emulate its behavior after recovery. During normal execution, our memory system simulator

logs memory requests (but not I/O requests) from Simics. If a recovery occurs, we stall Simics while replaying requests from our memory system logs. Since we do not handle I/O in our memory system simulator, we do not replay I/O requests. Thus, the simulation does not obey the rules for correctly handling the output commit problem. However, I/O is infrequent in our workloads, and we do not believe that its effect would be substantial.

Methodology. Commercial workloads running on real operating systems exhibit instability in their runtimes, and the simulation methodology must account for this effect or risk coming to incorrect conclusions. While our simulator is perfectly deterministic, even small perturbations of the workloads can cause wildly divergent execution paths, perhaps due to reorderings of lock acquisitions or operating system scheduling decisions. We account for this instability as outlined by Alameldeen et al. [4]. We simulate each design point multiple times with small, pseudo-random perturbations of memory latencies to cause alternative executions. When presenting performance results, we plot the mean values and error bars to represent one standard deviation in each direction (i.e., $\hat{\mu} \pm \hat{\sigma}$).

3.2.2 Workloads

Commercial applications are an important workload for high availability systems. As such, we evaluate *SafetyNet* with four commercial applications and one scientific application, for comparison. *SafetyNet* performance depends on the application, because applications have different workload intensities (i.e., frequencies of update-events). More intense workloads are more likely to fill up the CLBs and cause stalls, as shown in Equation 2 in Section 3.1.1. Conversely, to get equivalent performance for a more intense workload may require larger CLBs. Also, more intense workloads have greater rates of store overwrites, and this effect can degrade performance, as shown in Equation 4.

The worst-case workload for *SafetyNet* would have both a high frequency of update-events and poor locality (thus reducing the benefits of optimized logging). An example of such a workload would be a streaming multimedia application. Fortunately, these two factors are often inversely related. Poor locality leads to more cache misses, and cache misses incur latency that delays future update-actions. While systems may allow for multiple outstanding requests, this optimization does not completely alleviate the negative feedback loop due to cache misses.[3] Larger cache sizes may be used to reduce miss rates, so a workload like

3. Store latency also can be at least partially hidden by more relaxed memory consistency models, such as processor consistency (PC). As discussed in Section 2.1.6, implementations of PC hold the state of these stores in a store buffer that must be checkpointed by *SafetyNet*.

successive over-relaxation (SOR) that performs many writes and fits in the larger cache could exert more pressure on the CLBs. However, larger caches will make the CLBs look relatively smaller. Conversely, the CLBs could be made larger, to match the prior ratio of cache size to CLB size.

While this evaluation focuses on commercial workloads, other classes of workloads exist with other characteristics. Other workloads, such as data-intensive scientific applications, may stress *SafetyNet* more than commercial workloads. If these workloads are important for the system, *SafetyNet* implementations may want to reduce T_c, in order to reduce CLB pressure (as suggested by Equation 2). Moreover, an adaptive approach to setting T_c could be useful for systems that run a variety of workload types.

All of the workloads used in this evaluation are described in greater detail by Alameldeen et al. [4]. That paper addresses the setup, tuning, scaling, and warm-up of these workloads, as well as details about their performance characteristics. The workloads are all sized and warmed up to be memory-resident and avoid disk I/O. We now briefly describe each workload, and we characterize their execution behaviors (for a system unprotected by *SafetyNet*) in Table 3-2.

Online Transaction Processing (OLTP). Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 1 GB 10-warehouse database stored on five raw disks and an additional dedicated database log disk. There are eight simulated users per processor. We warm up for 10,000 transactions, and we run for 500 transactions.

Java Server. SPECjbb2000 is a server-side Java benchmark that models a 3-tier system and includes driver threads to generate transactions. We used Sun's HotSpot 1.4.0 Server Java Virtual Machine. Our experiments use 24 threads and 24 warehouses (~500 MB of data). We warm up for 100,000 transactions, and we run for 50,000 transactions.

Static Web Server. We use Apache 1.3.19 (`www.apache.org`) for SPARC/Solaris 8, configured to use pthread locks and minimal logging as the web server. We use SURGE [8] to generate web requests. We use a repository of 2,000 files (totalling roughly 50 MB). There are ten simulated users per processor. We warm up for 80,000 requests, and we run for 5,000 requests.

Dynamic Web Server. Slashcode is based on a dynamic web message posting system used by `slashdot.com`. We use Slashcode 2.0, Apache 1.3.20, and Apache's `mod_perl` 1.25 module for the web server. MySQL 3.23.39 is the database engine. The database is a snapshot of `slashcode.com`, and it contains 3,000 messages. A multithreaded driver simulates browsing and posting behavior for three users per processor. We warm up for 240 transactions, and we run for 50 transactions.

TABLE 3-2. Workload execution behavior

Workload	Dynamic Instruction Count	Instruction Footprint (Mbytes)	Data Footprint (Mbytes)	L1I Cache Misses/ 1000 Instrs	L1D Cache Misses/ 1000 Instrs	L2 Cache Misses/ 1000 Instrs
SpecJBB	3.7 billion	1.6	221	0.9	9.0	4.5
Apache	10.5 billion	1.1	84	1.1	2.8	2.4
Slashcode	7.3 billion	2.7	144	1.2	3.2	1.1
OLTP	8-10 billion	2.0	50	2.8	2.0	1.8
Barnes-Hut	11.4 billion	0.5	21	0.3	3.0	1.6

Scientific Application. We use *barnes-hut* from the SPLASH-2 suite [118], with the 64K body input set. This scientific application places much less stress on the memory system, and it serves as a comparison to the commercial workloads. We begin measurement of this application at the start of the parallel phase, in order to avoid measuring thread forking.

3.3 Experiments

We perform three experiments to evaluate *SafetyNet* performance, and we show their results in Figure 3-1. For each of our five workloads, we plot five bars: two bars for systems unprotected by *SafetyNet* and three bars for systems with *SafetyNet*.

3.3.1 Experiment 1: Error-Free Performance

In this experiment, we run two systems, *SafetyNet* and unprotected by *SafetyNet*, in a error-free environment. In Figure 3-1, the first and the third bars (from the left) for each workload reflect the normalized performances of the unprotected system and *SafetyNet*, respectively. We observe that the two systems perform statistically similarly for three out of the five workloads. For the other two workloads, jbb and slashcode, there is a small performance degradation. With 512-kbyte CLBs, stalls occur often enough in these workloads to impact performance. However, if CLBs are increased to 1-Mbyte, these workloads suffer no performance penalty with *SafetyNet*.[4]

Inspecting Equation 1 in Section 3.1.1 provides insight into the similarity between *SafetyNet* performance and the performance of the unprotected system. First, CLB stalls occur rarely, so they contribute little overhead. Second, overhead due to processor register checkpointing (100 cycles out of every 100,000 cycles) is

4. Mean performance results for OLTP incorrectly suggest that *SafetyNet* outperforms an unprotected system. Statistical examination of the results reveals that performance is comparable.

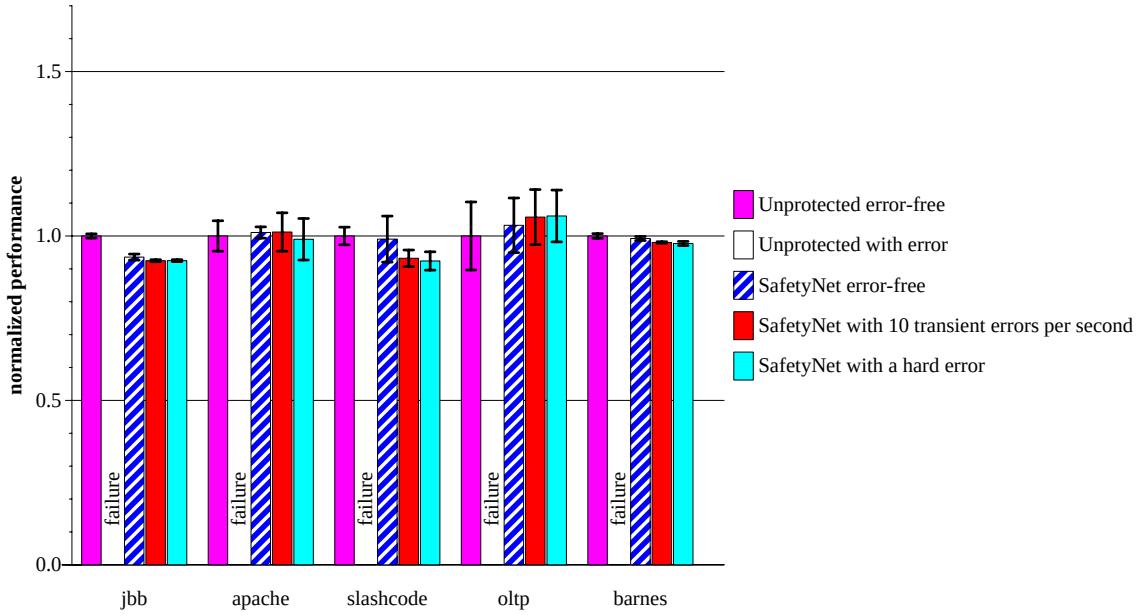

FIGURE 3-1. Performance comparison of *SafetyNet* with an unprotected system

within the noise. Third, overhead due to stores that require logging is negligible, since such stores comprise 0.1% of all instructions.

3.3.2 Experiment 2: Dropped Messages

In this experiment, we periodically inject transient errors into the system by dropping a message[5] every one hundred million cycles (i.e., ten times per second). The requestor times out on its request and triggers a system recovery. The second "bar" reflects the unprotected system performance (failure). The fourth bar from the right represents *SafetyNet* behavior, and we see that it is statistically similar to the error-free scenario.

Inspecting Equation 5 and Equation 6 in Section 3.1.2 helps to explain this result. Equation 5 shows that recovery cost is proportional to the error rate. The dominant cost of recoveries in Equation 6 is having to replay the lost work. Thus, the lack of performance impact is not surprising, since recovering even 400,000 cycles of work (i.e., the very worst case for a system with T_c=100,000 cycles and four checkpoint contexts) is a small fraction of the hundred million cycle error period (i.e., error period is the reciprocal of the error

5. We abstract the fault itself (e.g., a cosmic ray uncorrectably garbles a message) for generality.

rate). Moreover, the exact system recovery latency is not critical, since *SafetyNet*'s recovery latency is orders of magnitude shorter than the latency of failing and rebooting (while preserving data integrity).

In Section 3.4.4, we explore *SafetyNet*'s sensitivity to changing the frequency of soft errors, including error frequencies as high as one per ten million cycles. As the model would suggest, increasing the error rate will, at some point, cause a visible degradation in performance.

3.3.3 Experiment 3: Lost Switch

In this experiment, we inject a hard error into an interconnection network switch, killing a half-switch, after 5 million cycles.[6] Recall from Section 3.2.1, that each switch in the torus is comprised of two half-switches, which provides sufficient redundancy in the case that a single half-switch dies. However, the loss of the half-switch causes the loss of its buffered messages, which must be tolerated. Moreover, the system must reconfigure the interconnection network to route around the dead half-switch.

The second "bar" reflects the failure of the unprotected system. The fifth bar reflects *SafetyNet* performance, and we observe that, most importantly, *SafetyNet* avoids a failure. Its performance is slightly degraded, with respect to the error-free scenario, due to the restricted post-error bandwidth.[7]

Given a system with enough bandwidth not to be overly impacted by losing some of it due to a dead half-switch, which is the case in this experiment, the same discussion as in Section 3.1.1 explains why performance is not degraded. In a system that was more bandwidth-starved, the loss of bandwidth after the recovery would have a more pronounced effect, but here it has only a small impact on performance.

3.4 Sensitivity Analyses

In this section, we perform sensitivity analyses to gain a better understanding of how *SafetyNet* performs for different system parameters and different workload behaviors.

---

6. We do not model the diagnosis of this error. While we discuss diagnosis for this error earlier in the thesis, the details of modeling it are system-specific and do not help to illustrate this example.

7. We do not model the latency to reconfigure the interconnection network, because this latency is both difficult to estimate and a one-time penalty. The more important result is that the system does not fail and that the long-term performance of the system may be affected by the loss in interconnect bandwidth.

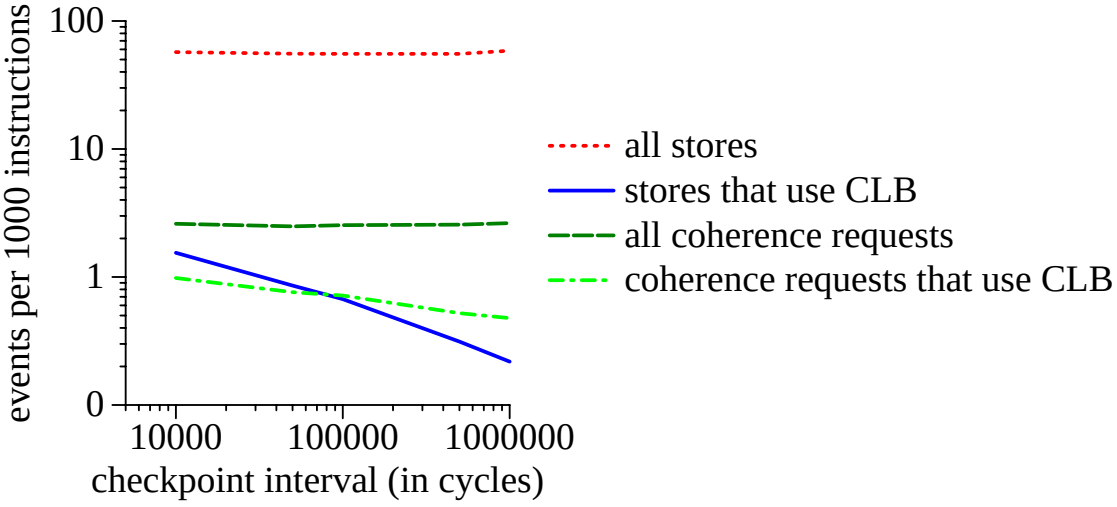

FIGURE 3-2. Workload intensity (Apache workload)

3.4.1 Checkpoint Log Buffer Storage Cost

As described in Section 2.2.3, an implementation of *SafetyNet* seeks to size the CLBs to avoid performance degradation due to full CLBs. Total CLB storage is proportional to the number of allowable checkpoint contexts and the number of entries per checkpoint. We allow for four checkpoints and a CLB entry is 72 bytes (8-byte address and 64-byte data block). The number of entries per checkpoint corresponds to the logging frequency which is, in turn, a function of the workload intensity. In Figure 3-2, for the static web server workload, we plot logging frequencies as a function of the checkpoint interval. We scale both the *x* and *y* axes logarithmically. Distinguishing between all stores/requests and only those stores/requests that require logging, we notice the striking drop-off in the latter as the checkpoint interval increases. These trends are the same for the other workloads, and the intuition explaining this phenomenon is that spatial and temporal locality reduce the number of distinct blocks touched per checkpoint interval. Figure 3-2 shows that, on average, only about 100-150 CLB entries are created per 100,000 instructions (although the variance in this rate requires more storage). Starting at intervals of 10,000 cycles, increasing the interval length by a factor of ten only increases CLB occupancy by a factor of between five and seven.

Equation 2 in Section 3.1.1 suggests that performance will improve (i.e., CLB stall time will decrease) as CLB size increases, but the relationship is non-linear. For example, at some CLB size, increasing CLB size further will have no effect. In Figure 3-3, we plot the performance of *SafetyNet* as a function of CLB size. While 1-Mbyte CLBs produce statistically comparable performances across the workloads, 512-byte

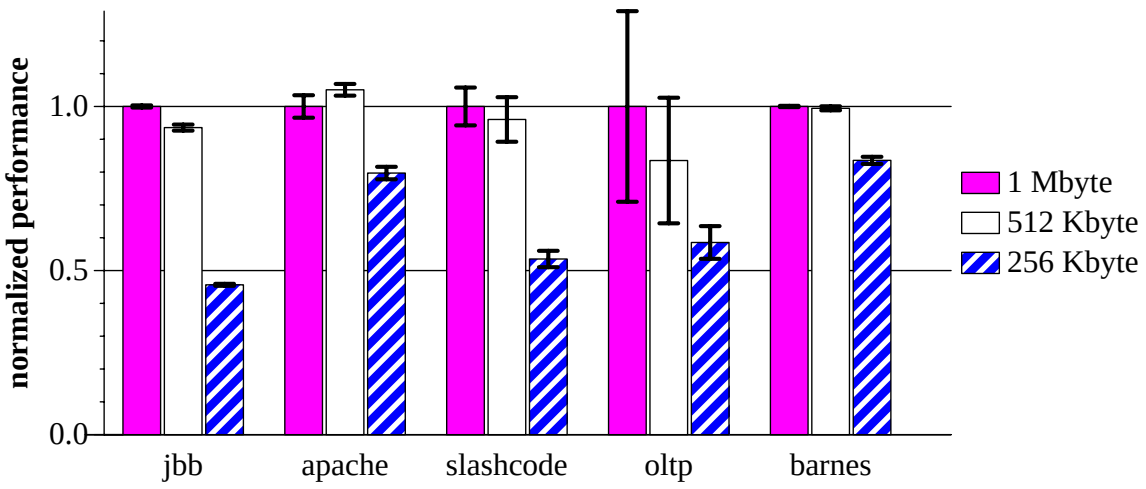

FIGURE 3-3. Performance vs. CLB size

CLBs slightly degrade the performances of two of our workloads, and 256-kbyte CLBs significantly degrade the performances of all of our workloads.

3.4.2 Checkpoint Interval Length

The checkpoint interval, T_c, is an important *SafetyNet* parameter for several reasons. First, a longer interval allows for greater tolerance of error detection latencies. Second, the checkpoint interval determines when a system with *SafetyNet* can interact with the outside world. Due to the output commit problem, discussed in Section 2.1.3, the system cannot send data outside its sphere of recoverability until it has been validated. Since validation latency depends on the checkpoint interval, the checkpoint interval thus plays an important role in I/O. For low-performance I/O, such as disks and external networks, delaying communication to wait for validation is a negligible cost. However, for high performance I/O, such as cluster communication, the maximum allowable checkpoint interval may be a function of the required I/O performance.

To gauge the impact of varying the checkpoint interval[8], we plot *SafetyNet* performance versus the checkpoint interval length in Figure 3-4, normalizing to the base case of 100,000 cycle intervals. The performance model in Section 3.1.1 suggests two trends. First, Equation 3 shows that shorter intervals will only hurt performance if they are so short that the 100 cycle register checkpointing overhead becomes signifi-

8. We do not study its affect upon the output commit penalty, since our workloads are scaled and warmed up so as to reside in memory and avoid disk I/O.

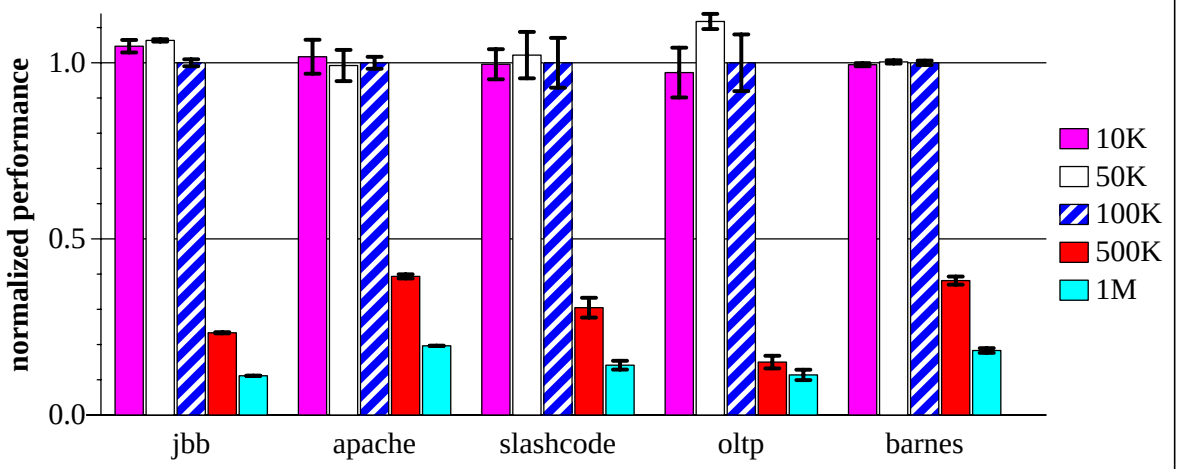

FIGURE 3-4. Performance as a function of checkpoint interval (512 kbyte CLBs)

cant. Second, Equation 2 shows that longer intervals will eventually start to degrade performance as CLB stalls become more frequent.

What we see in Figure 3-4 are two trends. First, smaller checkpoint intervals perform the best and perform comparably to each other, because 100 cycles is in the noise even for intervals as short as 10,000 cycles (i.e., 100 cycles is 1% of this interval length).[9] Our second observation about Figure 3-4 is that intervals of 500,000 cycles and longer perform significantly worse than smaller interval lengths. As checkpoint intervals lengthen, the pressure on the CLBs increases, since CLB usage is a function of interval length (and workload intensity). While the pressure does not increase linearly with interval length, due to optimized logging, the increase can be significant. Consider the extreme case in which one interval spans the entire execution—in this case, CLB usage (distributed across the system) is proportional to the memory image of the execution.

To further validate this hypothesis, we re-ran this experiment with larger CLBs and noticed striking improvements in performance. With large enough CLBs, we can attain the same performance as was achieved with smaller checkpoint intervals. In Figure 3-5 and Figure 3-6, we plot the performance of each workload with intervals of 500,000 cycles and one million cycles, respectively, as a function of CLB size. For the 500,000 cycle checkpoint intervals, CLBs of two Mbytes or greater appear sufficient to avoid performance degradation. There is a steep drop-off in performance, though, for CLBs less than this size. For the one million cycle checkpoint intervals, even two-MByte CLBs sacrifice considerable performance, as

9. OLTP’s performance advantage for 50K checkpoint intervals is a statistical anomaly.

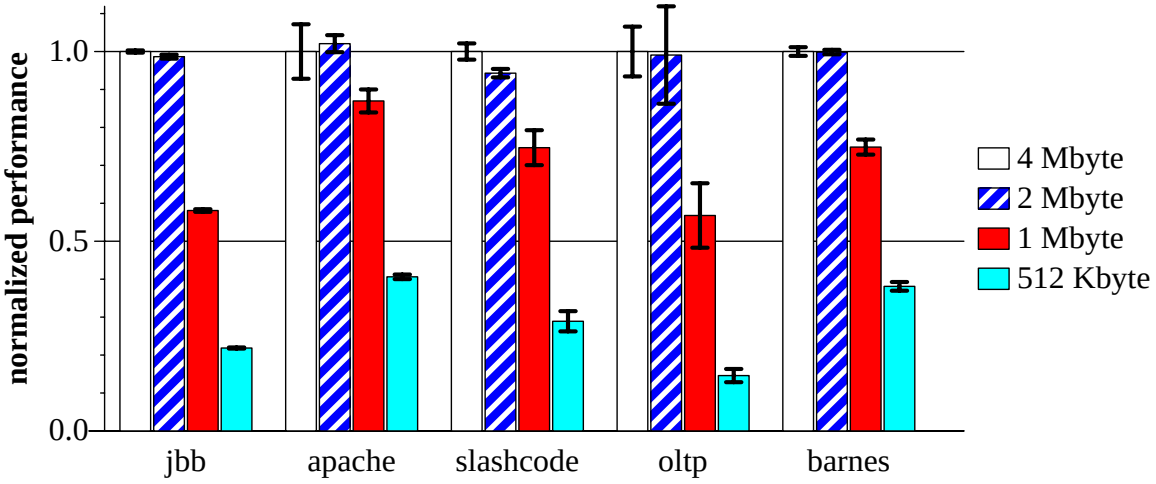

FIGURE 3-5. Performance vs. CLB size for 500,000 cycle intervals

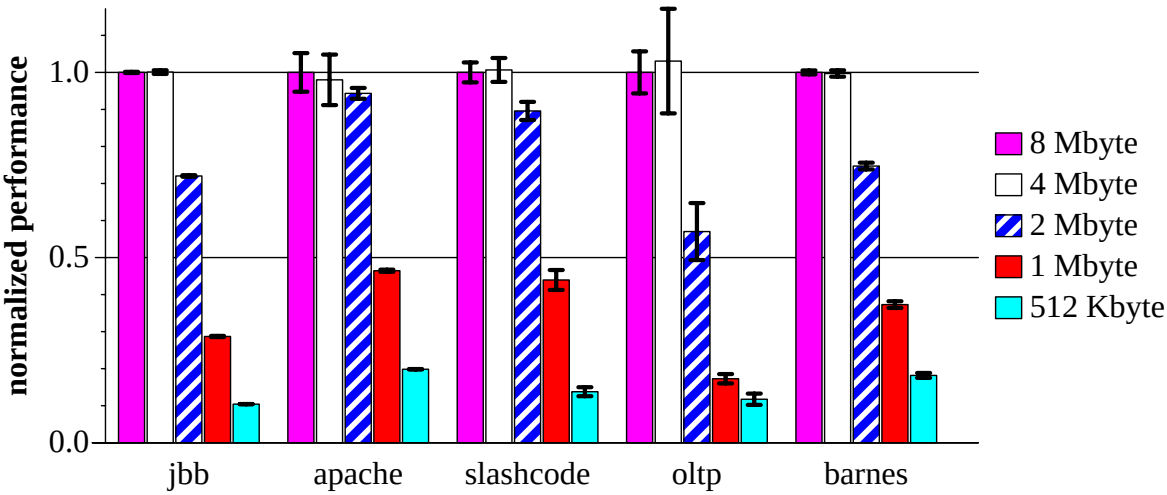

FIGURE 3-6. Performance vs. CLB size for 1 million cycle intervals

evidenced by the performance improvement achieved with four-MByte CLBs. However, four Mbytes appears sufficient in this case, since there is no performance gain evidenced by going to eight MBytes.

3.4.3 Register Checkpointing Latency

In Chapter 2, we discussed how processors must be able to checkpoint their register state whenever a new checkpoint is created. There are numerous ways to implement this capability, with varying trade-offs

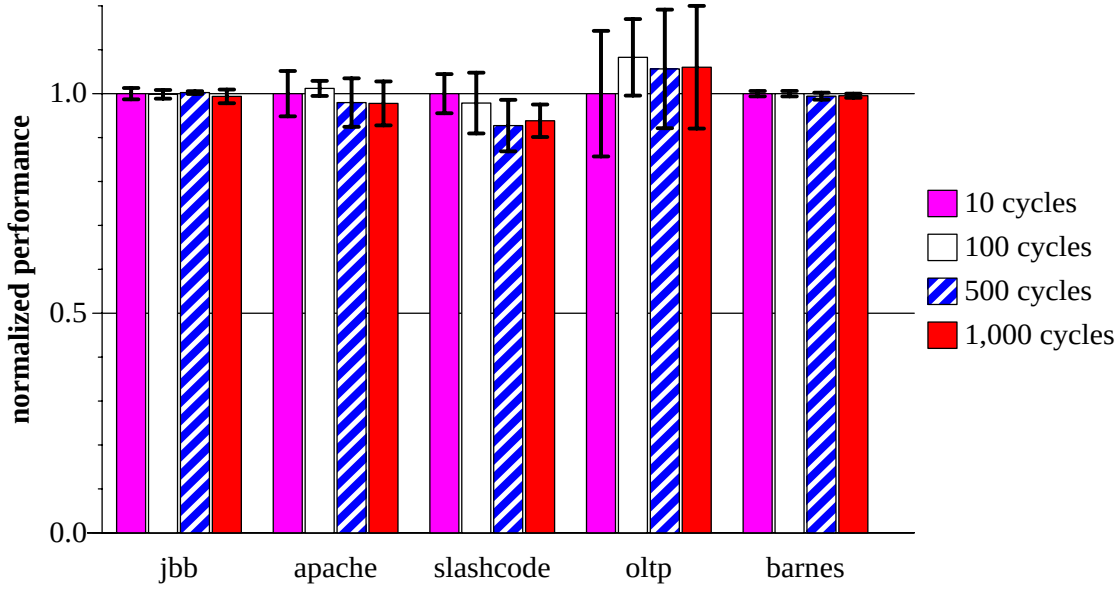

FIGURE 3-7. Performance as a function of register checkpointing latency

between speed and complexity. Designers only want to implement what is necessary, so we wish to understand how fast checkpointing must be so as not to perceptibly degrade performance.

To explore the impact of register checkpointing latency on *SafetyNet* performance, we plot performance as a function of register checkpointing latency in Figure 3-7. Equation 3 shows that this latency will only matter if it is an appreciable fraction of the checkpoint interval length, T_c. Unsurprisingly, we observe that register checkpoint latency has a negligible effect on performance for checkpoint intervals of 100,000 cycles. The infrequency of checkpoint creation causes this overhead to be lost in the noise. We conclude from these results that optimizing register checkpointing is not worthwhile, unless checkpointing is to be done far more frequently. Thus, the simple but not optimized solution presented in Chapter 2 will suffice.

3.4.4 Sensitivity to the Rate of Soft Errors

In Section 3.3.2, we demonstrated that *SafetyNet*'s performance in the presence of one soft error per hundred million cycles (i.e., ten times per second) was statistically equivalent to its performance in the absence of errors. In this section, we explore the performance impact of higher error rates. While these hardware error rates may seem exorbitant for today's technology, the trends are leading towards ever-increasing error rates, and it is instructive to test *SafetyNet*'s ability to keep up with these trends.

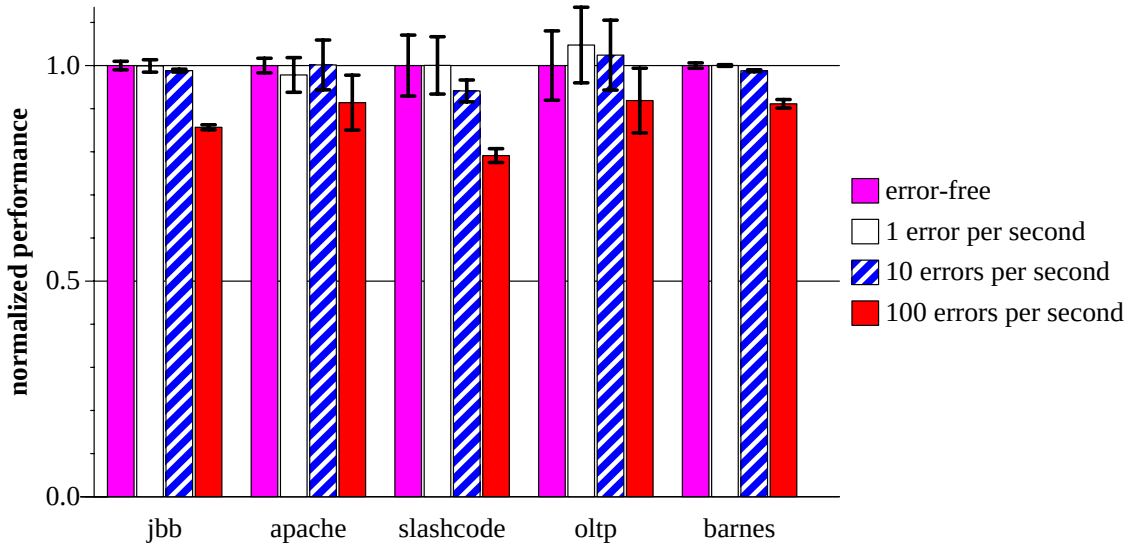

FIGURE 3-8. SafetyNet performance vs. soft error rate

In Figure 3-8, we graph *SafetyNet* performance as a function of the soft error rate. We perform this experiment similarly to the experiment in Section 3.3.2, by periodically dropping a cache coherence message. Equation 5 shows that performance will degrade as error rates increase. We observe that performance only begins to suffer, as compared to the error-free case, once soft error rates reach the rate of one hundred errors per second. At this rate, which is equal to one error every ten million cycles, the recovery time becomes non-negligible. This experiment uses a deterministic, periodic distribution of errors to provide a worst-case stress test. Any clustering of errors would improve *SafetyNet* performance by overlapping recovery latencies.

3.4.5 Cache Bandwidth

SafetyNet's additional consumption of cache bandwidth (i.e., bandwidth not used by an unprotected system) depends on the workload intensity, particularly the frequencies of stores that require logging. These stores consume additional cache bandwidth for reading out the old copy of the block. Logging due to transferring cache ownership, however, does not incur additional bandwidth, since the cache line must be read anyway. As shown in Figure 3-2, for the static web server workload and a checkpoint interval of 100,000 cycles, only 2-3% of stores (less than 0.1% of all instructions) require logging. In Figure 3-9, we plot the percentage of cache bandwidth used by cache hits, cache fills, responding to coherence requests, and log-

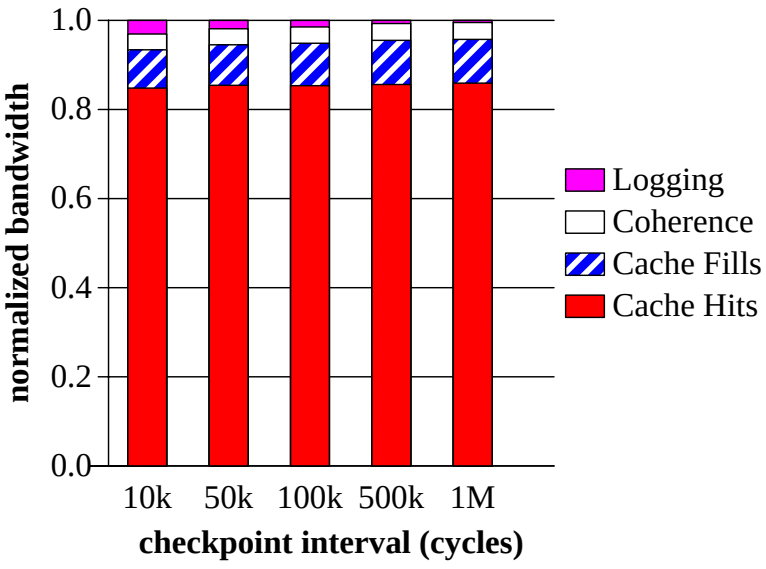

FIGURE 3-9. Bandwidth vs. checkpoint interval (static web workload)

ging due to store overwrites. The additional cache bandwidth used by *SafetyNet* ranges from 0.3% for million cycle intervals up to 4% for short 5,000 cycle intervals.

3.5 Summary

In this chapter, we have evaluated *SafetyNet*. We developed a qualitative analytical model for *SafetyNet* performance, and this model illustrates the system and workload parameters that influence system performance. We discussed our evaluation methodology, which incorporates full-system simulation and commercial workloads. We then quantitatively evaluated *SafetyNet* performance, comparing it to an unprotected system. We discovered that *SafetyNet* performs comparably to an unprotected system if the CLBs are sized appropriately. Moreover, *SafetyNet* avoids failures in the presence of hardware errors. Lastly, we explored several sensitivity analyses to determine the impact of varying system and workload parameters. Sensitivity analysis showed that *SafetyNet* performance is relatively robust, although it can be degraded significantly if the CLBs are not sized sufficiently.

Chapter 4

Availability

This chapter discusses how to improve system availability with *SafetyNet*. Given a checkpoint/recovery mechanism, the challenge of providing availability is reduced to the easier problem of error detection. *SafetyNet* can tolerate any device fault, provided that:

- The fault does not corrupt ECC-protected architectural state or other state that maintains the recovery point (e.g., CLB state).
- A system can be augmented with a mechanism to detect the resultant error (or determine its absence).
- The resultant error is detected while *SafetyNet* still maintains an error-free recovery point. An error that is not detected promptly can become latent and unrecoverable.
- The fault is transient or it is a permanent error that permits execution to resume after recovery, possibly after system reconfiguration.

We highlight a few example errors due to device faults in Table 4-1 (in which the first three rows comprise the first three rows of Table 1-1).

Numerous error detection schemes have been proposed in the literature and several have been used in practice. In the absence of a global recovery scheme, only localized error detection is applicable. Local error detection schemes, for the most part, are optimized for latency, since error detection is on the critical path. Thus, they often trade some detection rigor to avoid degradation of performance (e.g., using shorter error detecting codes).

SafetyNet can tolerate much longer error detection latencies, for two reasons. First, it is a global recovery scheme, so error detection is off the critical path for inter-component communication. With only local recovery, there is an output commit problem between components that forces error detection to be performed before communication. Second, *SafetyNet* pipelines checkpoint validation, and thus hides error detection latency equal to the product of the checkpoint period and the number of checkpoint contexts provided by the system. *SafetyNet* can maintain a recovery point as long as necessary, in the worst case, by stalling execution. However, error-free performance is best if, in the average case, error detection mecha-

TABLE 4-1. Classification of illustrative errors due to device faults[a]

	Error	Fault	Detection	Recoverable with SafetyNet	Resumability Mechanism
errors due to device faults	dead switch in ICN	hard device fault	timeout on request	yes	reconfiguration
	dropped coherence message	soft device fault	timeout on request	yes	none needed
	proc-cache chipkill	hard device fault	watchdog timer	no	not available
	bit flip on ICN link	soft device fault	error detecting code	yes	none needed
	bit flip in switch buffer	soft device fault	error detecting code	yes	none needed
	bit flip in CPU core	soft device fault	redundant thread	yes	none needed
	stuck bit in CPU	hard device fault	redundant thread	yes	not available

a. We shade the device faults that *SafetyNet* cannot tolerate.

nisms validate checkpoints error-free in one or a few checkpoint intervals (e.g, in 100,000 cycles or 0.1 milliseconds).

We begin in Section 4.1 by describing previously developed error detection techniques. In Section 4.2, we compare global recovery with *SafetyNet* to localized error recovery schemes. In Section 4.3, we develop innovations in error detection that are enabled by *SafetyNet*'s tolerance of long detection latencies. Most notably, we can innovate in hardware error detection by using long-latency, inter-node communication to periodically assert that certain system properties hold.

4.1 Traditional Hardware Error Detection Mechanisms

This section considers errors that *SafetyNet* could tolerate using traditional (i.e., localized and low-latency) error detection mechanisms. Section 4.1.1 to Section 4.1.5 focuses on errors due to faults in various parts of the system and how they can be detected. We do not focus on error diagnosis, since it does not significantly affect this discussion, but we will address diagnosis in later sections when it interacts with non-traditional uses of *SafetyNet*. We conclude in Section 4.1.6 with a discussion of errors that cannot be tolerated with *SafetyNet.*

4.1.1 Interconnection Network Errors

A typical interconnection network error model focuses on link errors, trying to detect single, double, or burst errors. Link errors are normally detected with error detecting codes (EDC), such as parity, single error correcting double error detecting (SECDED), or cyclic redundancy check (CRC) [30, 75, 80]. Current systems, such as the SGI Origin's Spider router [36], use short codes (e.g., on eight or sixteen bytes),

since the code must be checked before the data is forwarded or used. *SafetyNet* permits the use of longer and inherently stronger codes [76] because of its ability to tolerate long error detection latencies, as discussed in Section 4.2. Stronger codes may become more necessary for optical and RF interconnect technology, since these interconnects must use more power to achieve lower bit error rates [46].

SafetyNet is also compatible with other interconnection network error models. Lost and misrouted messages can be detected with time-outs. Time-out latency must be less than the error detection latency tolerated by *SafetyNet* (i.e., the checkpoint period times the number of checkpoint contexts), yet it should be long enough to avoid false positives (i.e., time-outs due to bad congestion). In our experiments, we choose a time-out latency that encompasses two or three congestion-free hops through the interconnect plus some slack for congestion. *SafetyNet* can also be used to recover from corrupted internal switch state (e.g., detected with internal EDC) and switch controller malfunction (e.g., detected with internal consistency checks).

4.1.2 Coherence Protocol Errors

There are numerous soft errors in the protocol engine that can be tolerated with global checkpoint/recovery. Transient faults in the protocol engine can produce errors such as sending the wrong message or sending duplicate messages, as well as errors in the reception of messages. Many of these errors are undetectable with error detecting codes, such as sending the wrong type of message or sending it twice. These messages are self-consistently correct and will not be flagged as errors by EDC even if they happen not to be the correct messages. These errors can, however, be detected by other mechanisms. An incorrect message or a duplicated message will always lead to an invalid transition at a coherence controller. For example, if a processor in a broadcast snooping protocol issues a Get-Shared instead of a Get-Exclusive, it will not be prepared to observe its own Get-Shared request when it arrives back on the address network. An invalid transition, such as the one in this example, will trigger a system recovery.

4.1.3 Cache Hierarchy and Memory Errors

Fault tolerance schemes for memory, both SRAM and DRAM, are already well-established, and we present the error model and prior detection techniques for completeness. A system with *SafetyNet* has to protect the cache hierarchy and memory with ECC, since they contain memory blocks that could potentially be the only valid copies in the system, so an uncorrectable error could be unrecoverable. *SafetyNet* might thus encourage stronger error correcting codes. Memory chip kills can be tolerated by using a RAID-like scheme for DRAM [28]. Unfortunately, a processor-cache chipkill partitions that chip's associ-

ated memory from the rest of the system, even if that memory itself is error-free. Tolerating this error model could be achieved with distributed parity techniques similar to those used by ReVive [82].

4.1.4 Processor Core Errors

Processor errors can be detected with numerous schemes. Re-computation with shifted operands [98] detects errors due to transient faults as well as some permanent faults. Redundant processors detect errors in high-availability systems, such as IBM mainframes [102] and Stratus machines [117]. These redundant processor schemes can tolerate processor faults, as well as detect their resultant errors, for a significant cost in replicated hardware. Recently, redundant threads have been used to detect transient processor errors. AR-SMT [91] was the first proposal, and subsequent research has further developed these ideas [87, 106, 69].

SafetyNet may not be the best solution for handling processor errors, since localized forward error recovery (FER) schemes can also tolerate processor faults. DIVA [6] implements dynamic verification of an aggressive processor core using a simple, verifiable checker core. Simultaneously and Redundantly Threaded processors with Recovery [110] tolerates processor faults using redundant threads. While these FER schemes could be used instead of *SafetyNet*, *SafetyNet* provides a unified mechanism that tolerates these faults, as well as others. Moreover, *SafetyNet* does not suffer a performance penalty due to the output commit problem between processors, whereas error detection and correction is on the critical path for localized FER schemes. With coordinated global recovery, nodes can exchange data with each other without having to first perform error correction, since the system can be recovered if an error is detected later. We explore this issue further in Section 4.2.

4.1.5 *SafetyNet* Hardware Errors

The *SafetyNet* hardware itself is also susceptible to faults, and we target single fault instances. We ensure that the service processor is not a single point of failure by using redundant service processors. Another possible single point of failure is the checkpoint clock in *SN-Directory*, so we distribute it redundantly. Most other faults in the *SafetyNet* hardware only manifest themselves during a recovery, which implies a double fault situation. While many double faults are tolerated by *SafetyNet*, a comprehensive coverage of them would require heavier-weight hardware support. One important error model that results from a form of double fault is the situation in which a cache or memory controller fails to log a block in the CLB and then another fault soon thereafter (i.e., before that unlogged block would have been deallocated) triggers a recovery.

Checkpoints of architectural state—processor registers, caches, CLBs, and memories—are protected with ECC, since an error in this state is unrecoverable if we have to restore a checkpoint. The CNs in the caches, however, cannot be protected with ECC, since a flash clear operates independently on a single bit of a word. Thus, we can protect the CNs by storing them redundantly. More efficient solutions may exist, but we do not pursue this issue further.

The mechanism for communicating messages regarding checkpointing (e.g., a message telling each node to validate a checkpoint) must tolerate faults. We assume a redundant transmission of these messages over the existing interconnection network. One possibility is time redundancy, in which a message is sent multiple times, possibly over different paths. Triple modular redundancy (TMR) with voting can mask a corrupted or lost message in any of the redundant transmissions. Performance is not critical for these messages, but reliable delivery is crucial.

4.1.6 Device Faults Not Tolerated with *SafetyNet*

SafetyNet, as currently specified, does not tolerate certain fault models. As previously discussed in Section 1.5.3, there are three primary reasons why a fault would be unrecoverable. First, if detection mechanisms do not exist to detect the resultant error, then the fault cannot be tolerated. While this may seem obvious, it is important to realize that an fault tolerance scheme is only as good as its corresponding error detection scheme. Second, if the fault corrupts the recovery point state, the system is unrecoverable. Such faults result in an error model that includes uncorrectable errors in architectural state as well as chipkill of a processor-cache chip. Third, a fault that prevents the resumption of execution after recovery is not tolerated. For example, a fault that partitioned the interconnection network would not be tolerated.

4.2 Global Recovery versus Local Recovery

In this section, we compare global recovery with *SafetyNet* and local error detection to localized error recovery schemes. Local recovery schemes may be simpler and faster than global recovery. However, they place error detection on the critical path, since there is an output commit problem for inter-component communication if global recovery is not available. Thus, error detection mechanisms must be fast. Global recovery takes error detection off the critical path, thus enabling longer latency error detection mechanisms. Global recovery exchanges the problem of local error *correction* to the simpler problem of local error *detection*.

In Section 4.2.1, we illustrate a general tradeoff between forward error recovery (FER), which is inherently a local recovery scheme, and global backward error recovery (BER). In Section 4.2.2, we illustrate two

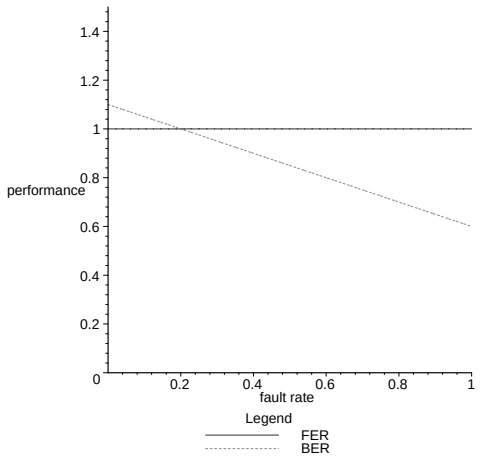

FIGURE 4-1. Rough comparison of BER vs. FER

We use a very simple linear model (with non-specific units) to compare BER and FER. FER runtime overhead is assumed to be 10% more than BER error-free performance. However, when errors occur, BER performance degrades by a constant recovery cost times the error rate. At the point where the two curves intersect, FER's runtime overhead equals BER's overhead due to error recoveries.

examples of trading local recovery for global recovery on links in the interconnection network. In Section 4.2.3, we discuss a similar tradeoff at the processors.

4.2.1 General Discussion of FER vs. Global BER

As a global checkpoint/recovery scheme, *SafetyNet* allows the system designer to trade FER for BER. Global BER avoids the output commit problem for communication between components. In some places where FER schemes are used to correct errors on transient or non-architectural (e.g., ECC on links), we can replace error correction with the inherently simpler task of error detection. Trading FER for BER, however, reveals a performance tradeoff that depends on error rates. If errors are frequent, FER is superior to a BER scheme which requires a costly recovery for each error. However, if errors are infrequent, BER is superior, since it optimizes the case of error-free execution and only pays a penalty when an error occurs. This tradeoff is illustrated with a very rough model in Figure 4-1.

4.2.2 Interconnect Link Errors

There are numerous local recovery schemes for tolerating faults that cause errors on links in the interconnection network. We discuss two examples of local recovery, one FER scheme and one BER scheme, and we compare them to using local error detection in conjunction with *SafetyNet.*

Local Recovery with Error Correcting Codes. Many systems use error correcting codes (ECC) to tolerate faults that cause bit errors on links. Since this state is non-architectural, we can trade ECC for EDC/*SafetyNet*. With global recovery, the system can detect (not correct) errors in the background, after having speculatively—in the sense that the data is predicted to be error-free—used or communicated the data. There are two advantages to this approach:

- Taking ECC off the critical path improves error-free performance and allows for longer latency error detection. This advantage is a specific case of BER's general advantage over FER for "low" error rates.
- Error detection is inherently easier than error correction. That is, an EDC code with a given number of redundant check bits, *k*, can detect more errors than an ECC code with *k* check bits will correct. For example, to detect *n* bit errors with a Hamming code requires a Hamming distance of *n+1*, while correction of n bit errors requires a Hamming distance of *2n+1*.

One caveat of using EDC instead of ECC is that we can only do this for non-architectural state. Thus, we must still use ECC to protect the trusted architectural state in the caches and memories (and in the processors' register checkpoint state, if that is not mapped to memory). With EDC instead of ECC, a fault in this state would produce a detectable error, but it would be unrecoverable and necessitate a system failure and reboot.

Another caveat of using EDC instead of ECC is that ECC is preferable if the error rate is sufficiently high. This issue is just a specific case of FER's general advantage over BER for high error rates. However, even then, a combination of the two schemes may be preferable. A short ECC masks the "easy" errors, while long EDC is performed in the background. For "tough" errors that ECC does not correct, EDC detects them and the system recovers with *SafetyNet*.

Local Recovery with Link-Level Retry. Besides EDC, other local recovery schemes exist for tolerating faults that cause link errors. The Spider router [36] in the SGI Origin uses link-level retry in the data link layer to handle errors on links. Link-level retry is a local BER scheme that only partially takes error detection off the critical path. Error detection does not slow down data link execution, but it must still be performed before data can be provided to the next highest layer in the protocol, the message layer. If, instead of link-level retry, we used EDC/*SafetyNet*, we would not have to wait for local error detection.

4.2.3 Processor Errors

Two choices for tolerating faults in a processor are to use either a recent scheme for localized FER or a combination of processor error detection and *SafetyNet*. DIVA [6] is a localized FER scheme for processors that uses a provably correct checker processor to gate possibly erroneous data from leaving an aggressive processor. The checker processor is on the critical path, but it does not significantly degrade error-free performance for quickly-detected errors.

Instead of using DIVA, we can use local error detection and *SafetyNet*. Recent processor error detection mechanisms, such as AR-SMT [91], have used redundant threads to detect errors due to transient faults. Combining AR-SMT with *SafetyNet* provides transient fault tolerance comparable to DIVA[1] without putting a checker processor on the critical path. Since DIVA's checker processor is not a major performance problem, the decision to use DIVA versus AR-SMT/*SafetyNet* depends mainly on other factors, including cost and complexity.

4.3 Innovations in Hardware Error Detection

Since *SafetyNet* is a global BER scheme that takes error detection off the critical path, we no longer need to optimize error detection for latency. Tolerating detection latency enables the use of error detection techniques that would otherwise be too costly in terms of performance. Most importantly, it allows for detection techniques that involve inter-node communication to determine whether end-to-end system properties are being maintained. The power of end-to-end error detection is appealing, if we can develop system-wide invariants that can be checked at a reasonable hardware cost.

Ideally, we would like to check that the system's memory consistency model is maintained, since consistency is the highest level of memory system correctness, but dynamic verification of memory consistency is a difficult problem (in fact, it is NP-complete in theory [38], although perhaps easier in practice) for future work. In this chapter, we will present two schemes for checking slightly lower-level system invariants. Verifying system-wide properties enables us to catch errors that are more difficult (or even impossible) to detect with localized error detection. Moreover, a higher-level error model can catch errors not specified in lower-level error models, and we will demonstrate examples of this property. However, a high-level error model does not necessarily help to diagnose the low-level error, similar to how a low-level error

1. DIVA also tolerates permanent faults and design faults in the aggressive processor.

model does not necessarily help to diagnose the fault that caused it.[2] Thus, if a high-level invariant fails repeatedly, the system must undertake a somewhat general diagnostic check.

In Section 4.3.1, we first discuss how to check system invariants, in general, with *signature analysis*. Components compute local signatures and perform a global reduction to detect errors. In Section 4.3.2, we sketch a simplified example of a signature analysis scheme. We then describe two realistic examples of signature analysis schemes. In Section 4.3.3, we describe how to check message-level invariants. In Section 4.3.4, we develop a technique for checking coherence-level invariants.

4.3.1 Detecting Errors with Signature Analysis

In this section, we describe in general how to use *signature analysis* to detect violations of system invariants due to errors. Signature analysis takes a large amount of input data—in this case, system states and events—and produces a small output, called a *signature*, that almost-uniquely characterizes the large amount of input data. The idea of signature analysis has existed for a long time, and it is widely used in built-in self-test (BIST) [1]. We will now discuss signature analysis, in general, before delving into the specifics of our signature analysis schemes.

All components in a signature analysis scheme maintain a local signature, *S(i)*, where *i* is the identity of the component. The local signature is updated for every event of interest, where the k^{th} event at component *i* is denoted *E(i,k)*. When obvious, we will denote an event simply as *E*, for clarity of notation. Signatures are updated according to an update function *U* that takes two parameters: *S(i)* and *E(i,k)*. Thus, *S(i) = U[S(i), E(i,k)]*. We assume that events are processed in order of occurrence. To check for errors, a global reduction of the local signatures is performed. The checking function, *C*, takes all of the local signatures as its variables, and produces a boolean result of the form *C[S(0), S(1), ... , S(N-1)] = {true, false}*, where *true* denotes that a error was detected.

There are certain properties that are desirable in *U*. The function *U* should be chosen so that the same signature almost never characterizes two different input streams (i.e., sequences of events), a phenomenon known as *aliasing*. We say that aliasing occurs if:

$$(S(i) = S(j)) \wedge \exists k | E(i,k) \neq E(j,k) \qquad \text{(EQ 7)}$$

2. A useful analogy is that a high-level error model is to a low-level error model what a low-level error model is to a fault model. A higher-level model can detect a wide range of lower-level faults, but it cannot diagnose them.

With perfect anti-aliasing in *U*, *C* will not miss any errors detected by the signature analysis scheme. In practice, however, engineering restrictions limit signatures to a finite number of bits, say *b*, and *b* bits can only represent 2^b sequences of events. Since systems will have more than 2^b possible sequences of events, different sequences will necessarily map to the same signature. A goal of signature analysis is to ensure that sequences that are almost identical do not map to the same signature, at the cost of allowing radically different sequences to map to the same signature.

In addition to anti-aliasing, *U* is also chosen to achieve certain properties that depend on the specific signature analysis scheme that is being used. For example, the signature analysis scheme in Section 4.3.3 will require *U* to be a commutative function, and the scheme in Section 4.3.4 will require a non-commutative function. A function *U* is considered commutative if:

$$U[U(S(i),E(i,m)),E(i,n)] = U[U(S(i),E(i,n)),E(i,m)] \quad \text{(EQ 8)}$$

To perform signature analysis in *SafetyNet*, all cache and memory controllers will maintain local signatures, update their local signatures with function *U*, and then the reduction check, *C*, will be performed at each checkpoint to detect errors. Thus, designing a signature analysis scheme entails choosing two functions, *U* and *C*. The reduction can be implemented on top of the existing mechanism for validating checkpoints. In Section 2.2.5, we explained how all cache and memory controllers send a message to the system service processor when they are ready to validate a checkpoint. We now add a signature as the payload of that message (i.e., component *i* sends *S(i)*), and the service processor performs the checking reduction *C*. If the check detects no errors (i.e., *C=false*), the service processor completes the validation by notifying every node. Otherwise, the service processor triggers a system recovery.

Signature analysis will detect the targeted errors unless one of three aliasing situations arises.

- Aliasing could occur due to finite resources for implementing the signature analysis scheme. We do not address this issue further in this thesis, since it is simply an engineering tradeoff, and this type of aliasing can be made arbitrarily small at the cost of additional hardware. Many typical signature analysis functions convolve the input stream with a pseudo-random number generator so as to reduce aliasing to an arbitrarily small probability. In hardware, pseudo-random number (PRN) generation is often implemented with a linear feedback shift register (LFSR) [40].
- Aliasing could occur because of a fault in the signature analysis hardware itself. This is a double fault scenario, and it could be tolerated with additional mechanisms, but we do not address this issue further.

- Aliasing could occur because the chosen update function, *U*, inherently suffers from aliasing. For example, if an update function adds the address of an incoming message to *S(i)* and the address is zero, then the occurrence of this event (the incoming message) is indistinguishable from the case in which it did not occur. As such, aliasing could occur even with infinite hardware resources for implementing the signature analysis. We address this form of aliasing in our examples of signature analysis, since it is a fundamental property of the schemes and not an implementation artifact.

4.3.2 Developing a Simplified Signature Analysis Example

In this section, we develop a simplified signature analysis scheme for purposes of illustration. The scheme is based on the "Kirchoff's Current Law" (KCL) reduction performed by the Thinking Machines CM-5 [62]. The CM-5 check ensured that the number of data messages entering any region of the interconnection network equaled the number of messages leaving the region. We simplify this invariant and check that the sending of each message has a corresponding reception. We assume that all messages have a single destination. An event, *E*, is the sending or reception of a data message. If sending corresponds to addition and reception corresponds to subtraction, a system-wide reduction should sum to zero.

The update function, U_{KCL}, is:

$$U_{KCL}[S(i), E] = \begin{pmatrix} S(i) + 1, \text{ if E is a send} \\ S(i) - 1, \text{ if E is a receive} \end{pmatrix} \quad \text{(EQ 9)}$$

Note that function arithmetic uses a finite number of bits with wraparound. The corresponding check function, C_{KCL}, is:

$$C_{KCL}[S(0),S(1),\ldots,S(N)] = \begin{pmatrix} true, \text{ if } \sum_i S(i) \neq 0 \\ false, \text{ otherwise} \end{pmatrix} \quad \text{(EQ 10)}$$

Aliasing can occur here for a variety of reasons. For example, if a double fault occurred such that one message was dropped in the network and another message was accidentally sent twice, this signature analysis scheme would not detect the error.

4.3.3 Checking Message-Level Invariants with Signature Analysis

In this section, we develop a signature analysis scheme for detecting message-level errors in *SN-Snooping*. We will detect all errors that lead to the loss, corruption, or reordering of messages in the snooping system.

We now develop a simplified update function, U_{ML}, for detecting message-level errors in the address network, and we will gradually describe a more sophisticated example. For each checkpoint that a cache or memory controller agrees to validate, it computes a signature based on the T_c coherence requests (i.e., address messages) it processed in that checkpoint interval. An event *E(i,k)* is the processing of the k^{th} incoming coherence request at component *i*. A simple update function, U_{ML}, adds the address of the coherence request, *Address(E)*, to the current value of the signature, *S(i)*.

$$U_{ML}[S(i), E] = S(i) + Address(E) \quad \text{(EQ 11)}$$

The check function, C_{ML}, detects if any component did not observe the same sequence of broadcasts as the rest of the components:

$$C_{ML}[S(0),S(1),\ldots,S(N)] = \begin{pmatrix} false, & \text{if } S(0) = S(1) = \ldots = S(N) \\ true, & \text{otherwise} \end{pmatrix} \quad \text{(EQ 12)}$$

Combining C_{ML} with this simple U_{ML} detects corrupted messages, some lost messages, and no reordered messages. First, we discuss aliasing that hides lost messages. Imagine the case in which a fault causes cache controller *i* to lose an incoming address message for address 22, and this was the $T_c{}^{th}$ message. Moreover, the $T_c{+}1^{th}$ message is also for address 22. At this point, cache controller *i* computes the "correct" signature and sends it to the service processor, and the error is not detected due to aliasing. A simple solution to this problem is to compute U_{ML} based on more fields of the message than just the address, such as the requestor (*Requestor(E)*) or request type, for example. We denote concatenation with a comma.

$$U_{ML}'[S(i), E] = S(i) + (Address(E), Requestor(E)) \quad \text{(EQ 13)}$$

The scheme described thus far still suffers from aliasing that may not detect reordered messages. In many broadcast snooping systems, the totally ordered address network is not implemented as a bus, but rather as a collection of buses [18] or a hierarchy of switches. In these interconnection networks, a fault can potentially lead to reordering of messages, which violates the required total order. An update function based on addition, which is commutative, will not detect these errors, since adding Message A before Message B produces the same signature as if they had been added in the other order. To avoid this form of aliasing requires a non-commutative function U_{ML}. An example of such a function is:

$$U_{ML}''[S(i), E] = (2 \times S(i)) + (Address(E), Requestor(E)) \quad \text{(EQ 14)}$$

So far, we have established two necessary qualities for U_{ML}:

- The input per message must be more than just the address, since otherwise repeated addresses can mask dropped messages.
- U_{ML} must be non-commutative, since otherwise re-ordered messages will not be detected.

The function that we choose, U_{ML}"", is a variant of U_{ML}" that is easier to implement in hardware. U_{ML}"" shifts *S(i)* one bit to the left (denoted by *S(i)* << *1*) and then Exclusive-ORs (XORs) the address and requestor of the incoming coherence request:

$$U_{ML}''''[S(i), E] = [S(i) \ll 1] \oplus [Address(E), Requestor(E)] \quad \text{(EQ 15)}$$

This function satisfies our two requirements and is also easy to implement in hardware. Similarly, we could have implemented a function using an LFSR, since signature analysis based on LFSRs is non-commutative and will therefore detect reordering errors, as well as corrupted or lost messages. LFSRs have better anti-aliasing properties (in terms of implementation-limited aliasing) than the function we chose. U_{ML}"" is intended more for illustrative purposes than as a final design point.

This signature analysis scheme avoids inherent aliasing, which enables it to reliably detect a wide class of errors. We detect all single instances of corrupted messages[3], dropped messages, and reordered messages. We detect many multiple fault situations, although any fault that affects the reception of the message at every node in the same way will elude detection.

We implemented this signature analysis detection scheme on top of *SN-Snooping*. To further test its capability to detect errors in this error model, we injected them into the system. The signature analysis scheme successfully detected the errors and triggered *SafetyNet* recoveries of the system. This signature analysis scheme catches some errors that are difficult to detect with localized error detection schemes. Most notably, it is difficult to detect in a broadcast snooping system if a node with shared permission to a block does not receive a Get-Exclusive for that block. In most directory protocols, lost messages are easy to detect because requests must be acknowledged. However, if this error is not detected in a broadcast snooping system and that shared node continues to load from the block, then a violation of coherence as well as the memory consistency model is possible.

The primary cost of this signature analysis scheme is extra hardware, since the latency of performing the signature analysis is hidden. Extra hardware is required to hold *S(i)*, but this hardware is simply a shift reg-

3. To be more precise, we detect all message corruptions in which the Address or Requestor field is corrupted. Detecting corruptions of other fields simply requires computing updates based on those fields, too.

ister. Hardware is also required for performing the update function, U_{ML}'". Since the signature is held in a shift register, computation of the new signature only requires XOR logic.

4.3.4 Checking Coherence-Level Invariants with Signature Analysis

In this section, we develop a scheme for testing coherence invariants in a cache coherence protocol. With update and check functions different from those in Section 4.3.3, we can check high-level invariants of the coherence protocol. For example, we can detect if a sharer did not downgrade permission to a block after an invalidation was received. Unlike the message-level scheme, we want a commutative function U_{CL}, since there are no ordering requirements for coherence. We just care that the coherence invariants are met within the checkpoint interval, but there are no ordering requirements within the interval itself.

The cache coherence invariant we choose to test is that every upgrade of coherence permissions at a controller (cache or memory) is reflected in an appropriate downgrade at one or more other controllers. This invariant is somewhat similar to an invariant that was statically checked off-line during the verification of the Alpha 21264 microprocessor [108]. At a high level, if an upgrade is considered an addition of a constant times the number of downgraders, and a downgrade is considered a subtraction of the same constant, the global reduction should sum to zero at the end of every checkpoint interval.

The check function, C_{CL}, is the same as C_{KCL} in Section 4.3.2, since this invariant is similar to the KCL invariant checked in that example:

$$C_{CL}[S(0),S(1),\ldots,S(N)] = \begin{pmatrix} true, \text{ if } \sum_i S(i) \neq 0 \\ false, \text{ otherwise} \end{pmatrix} \quad \text{(EQ 16)}$$

The update function, U_{CL}, operates on the address of the coherence request, *Address(E)*, i.e., *Address(E)* is the constant that is added/subtracted for each upgrade/downgrade event.[4] An upgrade adds *Address(E)* to *S(i)*, and a downgrade subtracts *Address(E)* from *S(i)*. For Get-Shared and Put-Exclusive requests, the update process is simple, since there is one upgrader and one downgrader. For a Get-Shared, the owner who satisfies the request is considered the downgrader. However, since a single Get-Exclusive upgrade can cause multiple downgrades, we would need to multiply the address that is to be added by the number of controllers that should downgrade as a result. There are different issues involved in implementing this analysis in snooping and directory coherence protocols, and we discuss both now.

4. To avoid aliasing due to situations in which *Address(E)=0*, we can use more sophisticated constants, such as *Address(E)* || *1*. To simplify notation, though, we simply use *Address(E)* in this discussion.

TABLE 4-2. Coherence-level signature update function (*SN-Snooping*)[a]

	Event *E*	Controller State	$U_{CL}[S(i), E]$
cache controller	Own Get-Shared	I->S	*S(i) + Address(E)*
	Own Get-Exclusive	I or S -> M	*S(i) + N*Address(E)*
	Own Put-Exclusive	O or M -> I	*S(i) - Address(E)*
	Other Get-Shared	I or S	*S(i)*
		O or M	*S(i) - Address(E)*
	Other Get-Exclusive	I, S, O, or M	*S(i) - Address(E)*
	Other Put-Exclusive	I, S, O, or M	*S(i)*
memory controller	Get-Shared	I or S	*S(i) - Address(E)*
		O or M	*S(i)*
	Get-Exclusive	I, S, O, or M	*S(i) - Address(E)*
	Put-Exclusive	O or M	*S(i) + Address(E)*
	address not at home		*S(i)*

a. Shaded entries reflect updates that have no effect on *S(i)*.

SN-Snooping. In *SN-Snooping*, the upgrader does not necessarily know how many downgraders exist. Ignoring the possibility of silent downgrades from Shared (Put-Shared requests), this problem can be solved by having the response from the owner to the upgrader (i.e., the data or the acknowledgment) include the number of downgraders, since the owner can keep track of this. However, most protocols do allow silent Put-Shared requests, so we solve this problem differently. Our solution to not knowing the number of downgraders for a Get-Exclusive is to assume that all other cache controllers and the home memory controller are downgraders. As shown in Table 4-2, a Get-Exclusive requestor increments by the request's address multiplied by the number of nodes in the system (*N*), and every other cache controller (*N-1*) plus the home memory controller decrements by the address (regardless of whether they have the block). A Get-Shared requestor increments by the address, and the owner who satisfies the Get-Shared request (either the memory controller or a cache controller in O or M) decrements by the address. A Put-Exclusive requestor decrements by the address and the memory controller increments by the address now that it is the owner. All of these transactions sum to zero if no errors occur, as shown in the equation for C_{CL}.

We implemented this coherence-level signature analysis error detection scheme on top of *SN-Snooping*. We injected errors into the system, including dropped messages and incorrectly processed messages, and the signature analysis indeed detected all of these errors. This signature analysis scheme, like the message-

level scheme presented in Section 4.3.3, can also detect errors that would be difficult to detect with more localized error detection schemes. If a Shared node processed an incoming Get-Exclusive from another node but did not invalidate its copy of the block, then the system can violate both cache coherence and the memory consistency model. This error example differs from the one in Section 4.3.3 in which the other node's Get-Exclusive was dropped before it could arrive at the Shared node.

The primary cost of this signature analysis scheme is also extra hardware, since the latency of performing the signature analysis is hidden. The current value of the signature is held in a register. Hardware is also required for re-computing the signature upon the arrival of every coherence request. The hardware to perform this computation is more complicated than that required for the message-level signature scheme, since the coherence-level signature function requires either subtraction or the combination of addition and multiplication. Thus, an adder and a multiplier are needed.

SN-Directory. While *SN-Directory* has less trouble than *SN-Snooping* with silent Put-Shared requests, it has two different problems. First, in three-hop transfers, a sharer that gets invalidated by a Forwarded-Get-Exclusive does not know the associated request's point of atomicity. In a three-hop transfer, the initial request is sent to the directory, which then forwards the request to the owner and sends invalidations to the sharers. The request's point of atomicity, however, is not determined until the forwarded request is processed by the owner. Second, it is difficult for a node to determine that it can send its signature for checkpoint CP to the service processor, since there could be Forwarded-Get-Exclusives in-flight towards that node. If these Forwarded-Get-Exclusives are part of a transaction with a point of atomicity in checkpoint CP, then the node would have needed to include them in its signature computation. Both of these problems can be solved with additional messages and complexity, but this brute force solution may not be worthwhile. Finding a more attractive solution is an open problem.

Summary of Coherence-Level Signature Analysis. This signature analysis scheme detects all single instances in which a transaction incurs a mis-matched number of coherence upgrades and downgrades. It also detects many multiple error situations, although not all. For example, a byzantine fault that caused an upgrader to downgrade and a downgrader to upgrade would cause an undetectable error.

4.4 Summary of Availability

In this chapter, we have addressed the issue of availability and how to improve it with *SafetyNet*. We first described some traditional error detection mechanisms and how they interact with *SafetyNet*. We then developed some innovative error detection techniques that are enabled by *SafetyNet*'s ability to tolerate

long detection latencies. Longer latency detection can be more powerful and more end-to-end (i.e., less localized) than schemes whose latencies are on the critical path. Notably, we developed two signature analysis schemes for detecting violations of system-wide invariants.

Chapter 5

Designability

Systems are becoming increasingly complicated, making both design and verification increasingly difficult. We would like to ease these problems by relying on *SafetyNet*, our checkpoint/recovery scheme, in the case of design errors, whether they are due to *speculatively correct design* or unintentional design faults. *SafetyNet* checkpoint/recovery unifies the support for designability with the support of availability that was discussed in Chapter 4.

In Section 5.1, we discuss speculatively correct design, and we discuss how to enable two examples of it with *SafetyNet*. In Section 5.2, we discuss how to use *SafetyNet* to tolerate certain classes of unintentional design faults.

5.1 Errors due to Speculatively Correct Design

Speculatively correct design of systems can improve performance and reduce costs, provided that errors due to mis-speculation can be detected and tolerated. The cornerstone of our philosophy is to allocate our resources—transistors, design time, verification effort—towards common-case events rather than rare corner-case events. If possible, we would like to reduce the cost of infrequent and complex events by simply treating them as “errors” and recovering from them. In the past, this approach has been employed for solving complex processor hardware problems in software. For example, processors have trapped to software for standard floating point arithmetic, such as the Intel 80386 without the 80387 floating point coprocessor. No recovery is necessary for these traps to software. Also, numerous architectures give the user the option of trapping to software for IEEE standard denormalized floating point arithmetic, including SPARC v9 [105] and Intel IA-64 [51]. Localized processor recovery may be necessary in these cases to support precise interrupt semantics. We seek to extend speculatively correct design beyond the processor and into the system. Since system components can thus communicate speculative data amongst themselves, we must provide global recovery to recover from these errors.

Errors due to speculatively correct system design fall into a specific region of the error space. In Table 5-1 (identical to the middle two rows of Table 1-1), we illustrate two examples of errors due to speculatively correct design. The cause of these errors (the fault) is mis-speculation, i.e., the designer intentionally did

TABLE 5-1. Classification of illustrative errors due to speculatively correct design

	Error	Fault	Detection/ Manifestation	Recoverable with SafetyNet	Resumability Mechanism
errors due to speculatively correct design	deadlock due to insufficient buffering in ICN (Section 5.1.1)	speculative underdesign	timeout on request	yes	slow-start execution after recovery
	out of order message arrivals on "in-order" ICN (Section 5.1.2)	speculative use of adaptive routing	invalid transition in protocol engine	yes	disable adaptive routing during re-execution

not design for certain circumstances, in effect predicting that these circumstances are rare. Detection is easier than in general because, by definition, the designer knows exactly where the faults are and how they manifest themselves as errors. Also, the errors that can arise due to the speculatively correct designs that we explore in this thesis are all recoverable with *SafetyNet*—otherwise, we could not employ speculation and maintain correctness in all situations. Lastly, speculatively correct designs must ensure resumability by avoiding livelock. Naively re-executing after recovery can lead to livelock, since the resumption of execution may keep encountering the same design error immediately after each recovery. For both examples of speculatively correct design that we present later in this section, we will explicitly describe our approach for avoiding livelock. We also address how to avoid high mis-speculation rates due to pathological situations.

In the rest of this section, we explore two examples of speculatively correct designs. First, in Section 5.1.1, we discuss how to simplify the design of deadlock avoidance in interconnection networks. Second, in Section 5.1.2, we enable adaptive routing in the interconnection network, even though the interconnect must guarantee point-to-point ordering of messages. Lastly, in Section 5.1.3, we discuss how to avoid pathologically bad mis-speculation rates.

5.1.1 Simplifying Deadlock Avoidance in Interconnection Network Design

Interconnection networks for multiprocessors are difficult to design, largely because it is difficult to achieve high and robust performance while verifying that deadlock is impossible under all situations. Interconnect deadlock (as opposed to coherence protocol deadlock) can occur due to the combination of cross-coupled requests and insufficient buffering for in-flight messages. For example, consider the simple example illustrated in Figure 5-1. In this example, switch S1 wants to send message M1 to switch S2, and S2

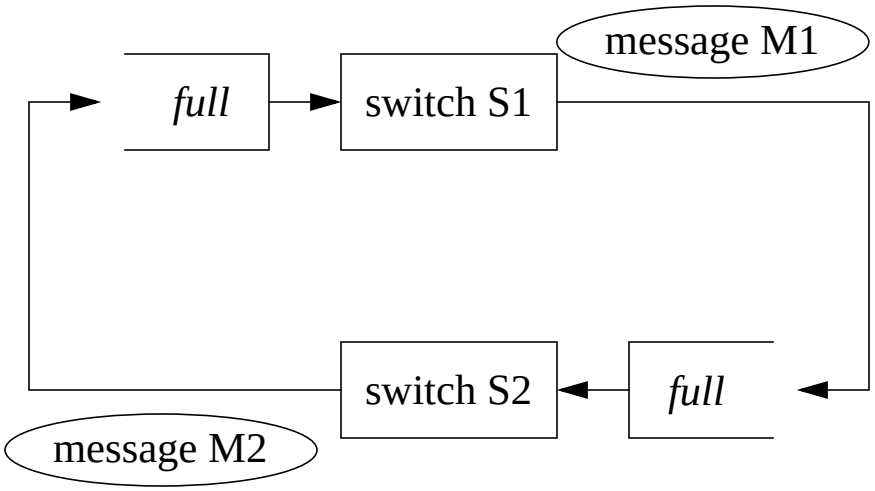

FIGURE 5-1. Example of deadlock in interconnection network

wants to send M2 to S1. However, the buffer from S1 to S2 and the buffer from S2 to S1 are both full and unable to accept new messages. Moreover, neither switch will process its incoming queue until it can send its outgoing message. Thus, if incoming message buffers are processed in FIFO order, the interconnection network is now deadlocked, since neither M1 nor M2 can make progress through the interconnect.

To avoid deadlock, interconnects either use worst-case buffering or some scheme, such as virtual channels [25], to break the cyclic dependences that can lead to deadlock. Employing worst-case buffering at each switch is the simplest solution, but the worst case is often far worse than the common case. Worst-case buffering can, for example, be proportional to the product of the number of nodes, the number of outstanding messages a node can have simultaneously, and the size of a message. Moreover, the worst case occurs exceedingly rarely, if at all, so we would like not to devote a disproportionate share of our resources to handling it.

To avoid the costs of worst-case buffering, a host of other techniques can be used to ensure that deadlock cannot occur. Flow control techniques restrict the flow of messages in the interconnect so that buffers are kept from filling up and potential deadlocks are avoided. The most common flow control technique is virtual channel flow control [25], a scheme that breaks the circular dependencies among messages that are necessary to get deadlock. Virtual channels break dependencies by assigning dependent messages on higher priority virtual channels. In our simple example, if M1 was on virtual channel 1 (VC1) and M2 was on VC2, then deadlock would have been avoided. Flow control techniques are well-understood, yet they are not simple to implement nor are they easy to verify correct. For example, the SGI Origin 2000 directory protocol [60] has only two virtual channels instead of the three (Request, Forwarded Request, and

Response) that would have ensured deadlock avoidance in all protocol situations. Instead, the Origin relies on a higher level mechanism to nack its way out of the deadlocks that occur due to this limitation. Given the expertise of the Origin's architects in this area, we do not suggest that this is a design error; rather, we use the Origin as an example of designability trade-offs. At the other extreme, the Alpha 21364 interconnect uses seven virtual channels [68], demonstrating that virtual channels are not prohibitively complicated.

Existing techniques for ensuring deadlock-free interconnects are either costly (worst-case buffering) or at least somewhat complicated (virtual network flow control). We would like to be able to design a simple network without resorting to worst-case buffering. The key to achieving this goal is to fall back on *SafetyNet* in those rare situations in which such an interconnection network deadlocks. We treat deadlocks as errors, similar to the device errors that were discussed in the context of availability.

Fortunately, the error model for this type of underdesign is clear and detection is straightforward. Deadlock in the interconnection network can be detected simply by time-outs at the requestor. Time-out latency is chosen to be long enough to mitigate false positives while short enough to be hidden by *SafetyNet*'s pipelined checkpoint validation. We set time-out latency equal to the sum of the latencies of three hops of congestion-free traversals through the interconnect, cache/memory access, and slack for congestion. If a message gets stuck in the network, the coherence transaction to which it belongs will not complete. The requestor of the transaction will timeout and trigger a system recovery. If time-outs are being detected repeatedly, the system performs diagnostics to determine the cause. If no switch or link is found to be dead, the system then assumes that the speculatively correct design is the culprit.

To avoid livelock, *SafetyNet* must ensure that the system will not continually deadlock due to insufficient interconnect buffering. Thus, the system temporarily enters a "slow-start" mode, in which nodes are only allowed to have one outstanding request. As long as we provide enough buffering in the interconnect to satisfy this restricted number of requests, slow-start provably avoids livelock.

To demonstrate the utility of easing interconnection network design, we implemented a network with buffering sufficient for the average case (i.e., less than worst-case buffering) and one virtual channel (i.e., the virtual channel is the physical channel). On top of this interconnection network, we run our MOSI directory protocol that normally requires four virtual channels to ensure deadlock avoidance.[1] In the case that deadlock is detected, the system recovers and resumes execution. If deadlock is frequent, the cost of recoveries will degrade performance, but infrequent deadlock will have negligible impact on performance.

1. Most directory protocols only require three virtual channels, but this protocol has a fourth for Final-Ack messages as explained in Chapter 2.

To determine the performance impact of such recoveries, we compare the performance of this system against a system with the same protocol running on an interconnection network with four virtual channels. We discover that the performance difference between the two systems is indistinguishable, for two reasons. First, deadlock does not occur. We can only get the system to deadlock when we reduce the buffering to the size of one entry per buffer and, even then, deadlocks are exceedingly rare. (Deadlocks are far more frequent when the memory system is driven by our random tester, which enables us to test our system more thoroughly.) Second, this experiment only measures performance loss due to deadlock recovery and not because of stalls due to limited buffering. Other buffers in the system, including those at the endpoints of the interconnect (i.e., where the interconnect meets the nodes) are infinite, so the system can drain unless deadlock occurs due to cross-coupled requests. Thus, buffers of size one perform equivalently to buffers of size five, except for the cost of the recoveries due to deadlock.

We conclude from this experiment that speculatively under-designing the buffering in the interconnection network is a viable solution to deadlock avoidance that allows the designer to target common-case execution. With *SafetyNet*, a designer can size the interconnect buffers for performance and not for correctness.

5.1.2 Enabling Adaptive Routing in the Interconnection Network

Interconnection networks can often achieve greater performance by using adaptive routing. Such interconnects allow, for example, two messages from switch S1 to switch S2 to take different paths. Adaptive routing can improve performance by distributing traffic more evenly across the interconnect and by enabling messages to be routed around localized congestion in the interconnect. In general, the flexibility of adaptive routing provides opportunities that system designers would like to be able to exploit.

A problem with adaptive routing, however, is that it complicates the enforcement of point-to-point order in the interconnection network. We illustrate this problem in Figure 5-2, in which two messages are sent from a source node to a destination node. The source sends message M2 after sending message M1, but M2 arrives first at the destination. The reversal in arrival order could be due, for example, to higher contention along the path taken by M1. With static routing, both messages would have followed the same path and thus arrived in order.

Some directory-based cache coherence protocols rely upon point-to-point order to avoid certain race conditions. One common example of these races occurs when the owner of a block, processor P1, sends a Put-Exclusive to the directory and another processor, P2, sends a Get-Exclusive for the same block to the directory that arrives first. The directory responds to both messages by sending a Put-Exclusive-Ack and a Forwarded-Get-Exclusive to P1. If those messages arrive in the reverse order of when they were sent (i.e., the

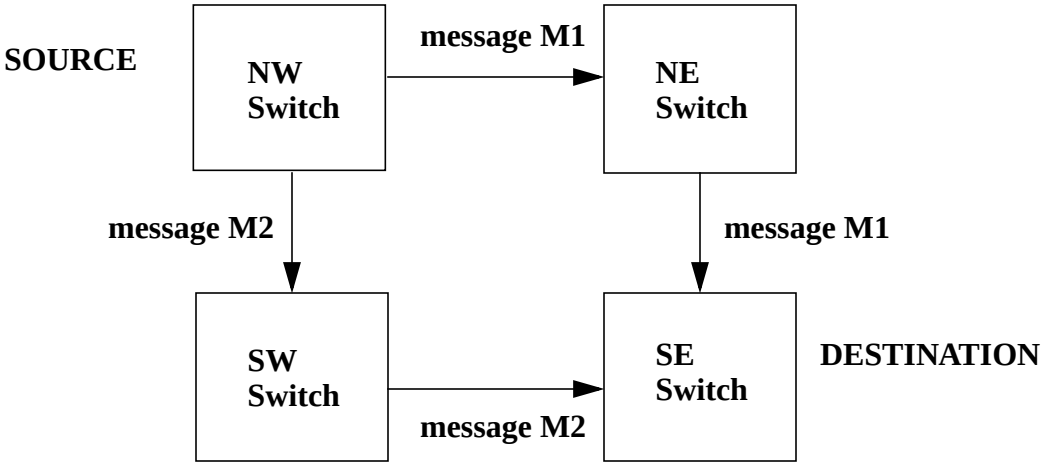

FIGURE 5-2. Violating point-to-point order with adaptive routing

Message M1 and message M2 are sent, in that order, from the NW Switch to the SE Switch. Message M2, however, arrives before message M1.

Put-Exclusive-Ack arrives first), then P1 sees the Put-Exclusive-Ack and downgrades to Invalid. Thus, it cannot handle the incoming Forwarded-Get-Exclusive.[2] Directory protocols can be designed to handle this race, but doing so complicates the protocol.

SafetyNet is well-suited to speculatively providing the illusion of a point-to-point ordered network in the presence of adaptive routing. First, the routing algorithm, while adaptive, is still unlikely to violate point-to-point ordering. Second, even when it does violate ordering, very few re-orderings impact correctness. Except in the example described above, re-ordering does not matter, for several reasons. First, in this protocol, point-to-point ordering is only required on one virtual network (the Forwarded Request virtual network). Second, ordering only matters for messages concerning the same block of memory. Third, even for messages concerning the same block, only certain messages need to be ordered. For example, multiple Forwarded-Get-Shared messages can be sent from the directory to the owner of a block, but the order in which they arrive does not matter for correctness. In particular, the situation in which a Put-Exclusive-Ack races a Forwarded-Get-Exclusive is particularly rare, since Put-Exclusive requests themselves are rare;

2. There is another common race case that is avoided by point-to-point ordering, although this case does not exist in our particular protocol. If the directory forwards a Get-Shared and then a Get-Exclusive for the same block to the owner, and the owner receives these forwarded requests out of order, then the owner will observe a Forwarded-Get-Shared in state Invalid (or some other incorrect state).

moreover, it is unlikely that a block that is being evicted at one node is actively wanted by another node. (This race happens much more frequently in our random tester than for real workloads.)

We implemented a *SafetyNet* system with an adaptively routed interconnection network and a directory cache coherence protocol that relies upon point-to-point ordering. The interconnection network, which is a two-dimensional torus, supports multicasts and broadcasts by splitting multicast/broadcast messages along their traversals. The adaptive routing algorithm allows messages (unicasts, multicasts, or broadcasts) to choose among minimal distance paths based on outgoing queue lengths in each direction. While adaptive routing can break point-to-point order, it can also cause deadlock. To isolate the issue of adaptive routing in this experiment, we avoid deadlock in this discussion by providing full buffering, although numerous more clever solutions exist, such as Duato's scheme for deadlock-free adaptive routing [29].

Classifying this error model in the error space clarifies the issues involved. Faults manifest themselves, as errors, as invalid transitions in coherence controllers, so we detect illegal message re-orderings by having cache controllers detect the specific incorrect transition in the coherence engine. For our race case, a cache with a block in state Invalid that receives a Forwarded-Get-Exclusive determines this situation to be an "error" and triggers a system recovery. This "fault" cannot manifest itself in any other fashion. Diagnostically, we assume that this situation arose because of a speculative re-ordering, even though it could be due to another cause/fault. We could use a more sophisticated diagnosis mechanism, possibly labeling messages with small sequence numbers. However, the simple solution which leads to occasional false positives (i.e., situations in which a cache controller receives a Forwarded-Get-Exclusive in Invalid for reasons other than speculative re-ordering), is sufficient.

This system appears to have more errors than a similar system without adaptive routing. To ensure resumability, we allow the interconnection network to disable adaptive routing temporarily, so that forward progress can always be made. Our heuristic disables adaptive routing until the resumption of execution has progressed beyond the point at which the error occurred that triggered the recovery.

We evaluated the performance of this system to determine if the positive benefits of adaptive routing outweigh the performance cost of recoveries due to illegal re-orderings. In Figure 5-3, we plot the relative performances of three systems with link bandwidths of 400 MBytes/second:

- Unprotected with static routing
- *SafetyNet* with static routing
- *SafetyNet* with adaptive routing

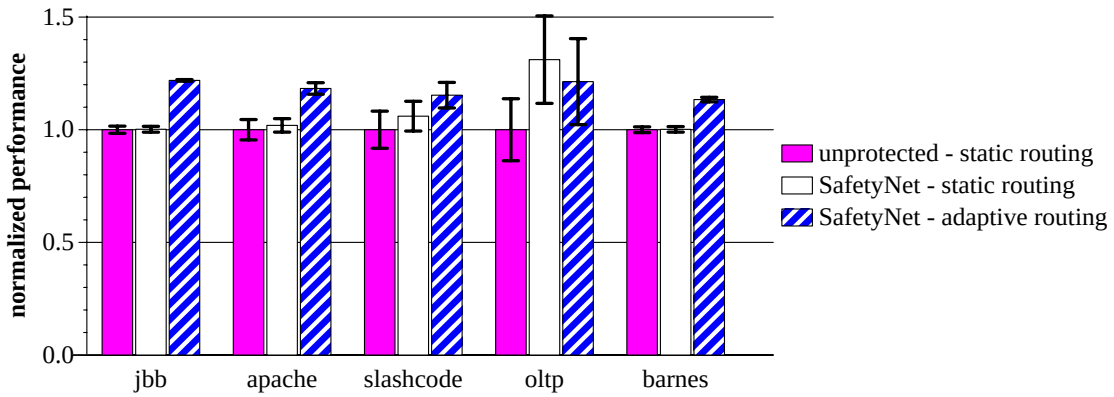

FIGURE 5-3. Performance of a system with adaptive routing

***SafetyNet* provides the illusion of a point-to-point ordered interconnection network.**

The adaptive routing scheme used in this experiment allows switches to route a message along any minimal-length path to its destination. Among these choices, the switch chooses the outgoing link with the smallest buffer occupancy.

We normalize the results to the performance of the unprotected system, and we observe that adaptive routing achieves a significant speedup for our workloads. There are two reasons for this speedup. First, the adaptive routing enables the system to better utilize its links. Link utilization was greater for every link with adaptive routing. Second, adaptive routing incurs very few recoveries, despite frequent re-orderings, because the vast majority of re-orderings do not affect correctness. In fact, we only observed a handful of recoveries in all of our simulations! However, we reassure ourselves that this race is possible and must be handled correctly, since it does occur occasionally in our workloads and it occurs much more frequently when we drive our system with a random tester instead of real workloads.

The performance impact of adaptive routing is partially a function of the available bandwidth provided by the interconnection network. With less bandwidth, adaptive routing has more opportunity to route around congestion. In the results just shown, we decreased the link bandwidth to 400 Mbytes/second, which is far less than the 6,400 MBytes/second assumed in Chapter 3. While this example may thus appear somewhat contrived, this experiment still provides a proof of concept. *SafetyNet* enables adaptive routing in a situation in which it was not previously possible.

5.1.3 Avoiding Pathological Mis-speculation

Speculatively correct designs will mis-speculate. The key is to ensure that mis-speculation rates cannot severely degrade performance. Mis-speculation rates must not rise too quickly as system parameters (e.g., processor speed, number of outstanding requests) or workload characteristics change. Both of these trends could increase the workload intensity and could cause more mis-speculations. Mis-speculation must also not fall victim to a pathologically bad situation, whether unintentional or due to malicious software.

Our two examples of speculatively correct design have both a natural feedback mechanism for reducing mis-speculation and fail-safe mechanisms for ensuring forward progress in pathologically bad situations. First, the natural feedback loop is the latency through the memory system. As workload intensity increases, due to system or workload trends, the memory system bottleneck limits the throughput of cache coherence transactions. Since these are closed systems, the feedback mechanism reins in even the most intense offered loads.

Second, both speculatively correct designs have a fail-safe mechanism for ensuring forward progress, even in the most pathologically bad situations. For the under-designed buffering example, the system can enter a slow-start mode for which the limited buffering satisfies the worst-case offered load in slow-start. For the adaptive routing example, the adaptivity can be temporarily disabled.

5.2 Errors Due to Unintentional Design Faults

SafetyNet can be used to tolerate unintentional design faults, if their resultant errors can be detected reliably [16, 34] and they permit resumption of execution after recover. The ability to tolerate unintentional design faults that slip through testing and verification could speed up a system's time to market. However, since these types of design faults are unintentional, it is difficult to target them with specific error detection mechanisms. By definition, these faults are not included in the fault model. Ironically, if we knew the fault model *a priori*, we would have avoided the design fault in the first place!

Unintentional design faults manifest themselves as errors at some point either late in the design/verification cycle or even after the system has been shipped. At this point, the producer has several unappealing options:

- Re-design the system to eliminate the fault and re-verify the system, possibly after having to recall the shipped product.
- Publish the existence of the fault.
- Ignore the fault.

TABLE 5-2. Classification of illustrative errors due to unintentional design faults[a]

	Error	Fault	Detection	Recoverable with SafetyNet	Resumability Mechanism
errors due to unintentional design faults	unspecified edge case in coherence protocol	unintentional design fault	invalid state in protocol engine	yes	slow-start execution after recovery
	Intel's FDIV bug [13]	unintentional design fault	self-checking program	yes	software FP routine
	routing bug in half-switch	unintentional design fault	invalid state in protocol engine or timeout	yes	*depends on specific bug*
	deadlock situation in coherence protocol	unintentional design fault	timeout at requestor	yes	*depends on specific deadlock*

a. We shade the errors that *SafetyNet* cannot tolerate (or may not be able to tolerate) without software support.

It would be preferable to instead tolerate errors due to the fault with *SafetyNet* and then perhaps re-design and re-verify a future spin of the system.

Classifying errors due to unintentional faults, which we will refer to as *design errors*, in the error space helps to focus on the difficult issues in tolerating these faults. We illustrate examples of design errors in Table 5-2 (in which the first two rows are identical to the last two rows in Table 1-1). The cause of these errors is design faults that did not get caught by testing or verification. Detection of design errors is difficult in general, but fortunately some types of design faults manifest themselves in ways similar to device faults. For example, a device fault that corrupts a coherence message by changing a Get-Shared request to a Get-Exclusive request looks much like a design fault that leads to a cache controller issuing a Get-Exclusive instead of a Get-Shared. *SafetyNet* can recover from some design faults, such as Intel's FDIV bug, that may require software intervention to avoid subsequent livelock in this case. Other types of design faults are simply unrecoverable.

For those unintentional design faults that manifest themselves equivalently to device faults, the error detection techniques presented in Chapter 4 suffice for detecting both classes of errors. Particularly for design errors, the end-to-end signature analysis detection methods will be effective. More localized error detection techniques are less able to detect errors due to system-level design faults. For example, link-level ECC in the interconnection network will not detect that a Get-Exclusive message was sent instead of a Get-Shared message, since the erroneous Get-Exclusive message will pass ECC. However, the coherence signature analysis technique presented in Section 4.3.4 will detect this design error. Another option for detecting design errors might be a field programmable detection mechanism for targeting faults in the field.

Diagnosis of an unintentional design error is a difficult challenge. When it is first detected, the designers and verifiers must try to reproduce it and determine its cause. The FDIV bug was diagnosed after reports from users that floating point division occasionally produced incorrect results. After initial diagnosis by the producer, the designers must determine whether the system can diagnose this error in the field and handle it appropriately. For the FDIV bug, this was not the case. For the design error that will be presented in Section 5.2.1, which is due to an unspecified corner case in a cache coherence protocol, the system can diagnose the error based on the specific invalid transition in a protocol engine, and it can take appropriate measures to ensure resumability.

In the rest of this section, we first present an example of an unintentional design fault that can be tolerated with *SafetyNet*, and then we present a more general discussion of which design faults can be tolerated.

5.2.1 An Example in the Cache Coherence Protocol

Cache coherence protocols define the behaviors of the cache and memory controllers. Each controller is a finite state machine (FSM) that has some number of states (per cache block) and handles some number of events that can happen to a block. Numerous controllers concurrently interact with each other with respect to many different blocks. While protocols are simple at a high level, they are much more complicated to design at a low level. Textbooks often abstract protocols into a handful of stable states (MOESI) and a handful of messages that are exchanged (in the easiest order for the reader to understand!) [24, 47]. In reality, though, protocols have numerous transient states, and messages race with each other in the interconnect and can arrive in many different orders.

Cache coherence protocols are notoriously difficult to design and (statically) verify. The state space explosion problem—an exponential function of the number of controllers, memory blocks, and block states—limits the effectiveness of formal verification methods [20], such as model checking and theorem proving. Testing is a valuable complement to formal verification techniques. Directed testing or random testing [9, 119] can uncover many bugs. Unfortunately, the complexity of coherence protocols is often due to subtle race conditions, especially those that are infrequent and thus less likely to be uncovered during testing.

We now present an example of a protocol race in *SN-Snooping* that the designer (the author!) did not initially consider. The designer overlooked this case until weeks later when random testing happened to uncover it (by crashing the simulator). We explore the potential to simplify coherence protocol design by treating this edge case as an “error” that triggers system recovery. As with the examples of speculatively correct designs in Section 5.1, the frequency of these now-allowable errors determines the viability of the speculation. If this protocol situation occurs more than rarely, the performance degradation due to recover-

ies could negatively impact performance. However, if we only recover in rare corner cases, then the impact of the infrequent recoveries should be negligible.

To test this hypothesis, we developed a version of the *SN-Snooping* protocol that treats a certain situation as an "error" instead of handling it. The situation arises when a cache controller has a block in state Modified (or Owned) and then issues a Put-Exclusive for the block, transitioning to a transient state. In this transient state, a Get-Exclusive arrives from another node, causing the cache controller to transition to a different transient state. Then, in this second transient state, the cache controller observes another Get-Exclusive from another node. This sequence of events is exceedingly unlikely, especially since it originates with a writeback from the cache controller. Compounding its rarity is that a block that is evicted by a writeback is unlikely to be requested by two other nodes. Moreover, both nodes must request exclusive access to the block in the interval of time between when the cache controller issues its Put-Exclusive and then observes its own Put-Exclusive on the address network. While this scenario is unlikely, it can occur. Our random tester took a long while to uncover this edge case, but it did occur. Thus, we must handle it appropriately.

In the error space, errors due to encountering a coherence transition that was not specified manifest themselves as invalid transitions. In this particular example, a cache controller that observes another node's Get-Exclusive while in the transient state described above triggers a system recovery. In the random tester, this is sufficient to preserve correctness. It may appear that no resumability mechanism is needed, since this error only occurs due to a timing race and the timing after recovery should be different than the timing that led up to the recovery. However, to *ensure* resumability, we must ensure a different timing. Thus, we temporarily enter a "slow-start" mode, in which nodes are only allowed to have one outstanding request.

We then tested the protocol on our set of commercial workloads, and all of them ran to completion without needing to recover from reaching the edge case. Thus, performance of the protocol is, for these workloads, identical to that of the fully designed protocol. While this obviously does not guarantee that the under-designed protocol will never have to recover, it does suggest the infrequency of recoveries due to encountering this corner case in the cache coherence protocol.

We conclude from this experiment that we can tolerate an unintentional design error in the cache coherence protocol with *SafetyNet*. Even if recoveries due to this error slightly degrade performance, the reduced time for design and verification provides a gain in performance (due to Moore's Law) that is likely to more than offset the cost of recoveries.

5.2.2 General Properties

While *SafetyNet* can tolerate the design fault example described in Section 5.2.1, this is by no means a general solution to design faults. Tolerating an unintentional design fault requires three properties:

1. Detection: The system can detect the error caused by the design fault.
2. Recoverability: *SafetyNet* can recover from this error model.
3. Resumability: *SafetyNet* can resume execution (without livelock) after recovering.

Currently, achieving any of these three properties is probabilistic. Moreover, the probability of achieving the second property is difficult to improve. However, we can improve the probabilities of the other two properties.

Detection. The probability of detecting design errors can be improved by adding better error detection capabilities to the system. Detection of design errors happens when detection mechanisms that target other error models also detect the manifestations of design faults. Checking for other error models helps to detect design faults that manifest themselves in similar ways to these newly detectable faults. Thus, we encourage the use of stronger error detection schemes.

Stronger error detection can be achieved with higher-level error detection mechanisms, because higher-level error detection can detect errors that are not in the low level error model. Higher-level error detection can be performed in hardware and in software. In hardware, end-to-end checking of high level invariants, such as the signature analysis schemes presented in Section 4.3.3 and Section 4.3.4, can detect numerous lower level errors. In software, self-checking programs [12] can similarly detect a wide range of errors, including design errors like Intel's FDIV bug.

Diagnosis is a component of detection that we would also like to improve, although this is certainly a difficult problem. We encourage the use of better diagnostic mechanisms, such as hardware instrumentation and system software diagnostics. Without diagnosis, the system cannot decide what action to take, if any, upon detection of an error.

Resumability. The probability of being able to resume execution can be improved by adding mechanisms that change an execution after a recovery. One example used thus far in this thesis is a slow-start execution mode, in which nodes issue requests at a slower rate. Slow-start enables a different, but still correct, execution by changing the timing in the multiprocessor system. Another example was developed in Section 5.1.2, where we described how to turn off adaptive routing after recovery.

The ability to dynamically turn off system features can help to avoid livelocks. Many industrial systems have followed this philosophy in order to more quickly ship a functional, if not fully-utilized, system. For example, Sun Microsystems shipped UltraSparcIII processors with hardware prefetching of floating point data, but they disabled this prefetching when it was discovered to be faulty [90]. In a later revision of the UltraSparcIII, the design fault was fixed and hardware prefetching of floating point data was enabled.

In general, more adaptive and more flexible systems are more likely to tolerate unintentional design faults. Field upgradable systems may also help in this regard. Since adaptivity and flexibility are also desirable for other reasons, we argue for designing systems with these properties.

5.3 Summary of Designability

In this chapter, we have addressed the problem of system designability. We discussed how to use *SafetyNet* to enable speculatively correct designs. Speculative correctness allows the designer to allocate resources towards the common case scenarios while falling back on *SafetyNet* for rare, unimportant cases. Lastly, we addressed the issues involved in using *SafetyNet* to tolerate unintentional design faults. While the ability to tolerate a fault for which the designer did not plan is probabilistic, we describe several avenues for improving the probabilities in this area.

Chapter 6

Related Work

SafetyNet is related to research in a number of different areas. Most of the related research addresses availability in the presence of hardware errors (Section 6.1), although some recent work has begun to address designability (Section 6.2). There also exists related research in checkpoint/recovery or versioning of data for use in other domains of computer science (Section 6.3). Finally, we discuss related work in using logical time to order events in distributed systems (Section 6.4).

6.1 Availability

Prior work in availability can be classified into two broad categories: backward error recovery (BER) through checkpointing or logging and forward error recovery (FER) through redundant hardware. We further distinguish BER schemes by whether they are implemented in hardware, software, or message-passing systems.

6.1.1 Hardware Backward Error Recovery

In BER schemes, the state of the system is checkpointed periodically or differences are logged. An error is tolerated by recovering to a previously checkpointed state or unrolling the log. IBM mainframes [44, 102, 96], which have been the archetypal high availability systems, have long used register checkpoint hardware and store-through caches to recover from processor and memory system errors, respectively. To tolerate some of the latency of error detection, Tamir and Tremblay [107] developed a micro rollback scheme that allows the recipient of erroneous information to rollback several clock cycles (which is the length of the window of opportunity for receiving erroneous information). *SafetyNet* differs from these approaches by tolerating hundreds of thousands of cycles of error detection latency.

Hardware BER schemes have often utilized the caches and/or the cache coherence protocol. The Cache-Aided Rollback Error Recovery (CARER) scheme [48] for uniprocessors uses a normal cache with a writeback update policy to assist rapid rollback recovery. This scheme is integrated with the cache controller, checkpointed system state is maintained in main memory, and checkpoints are established whenever a modified cache block needs to be replaced. Ahmed et al. [2] extend CARER for multiprocessors by syn-

chronizing the processors whenever any of them need to take a checkpoint. Wu et al.'s [120] multiprocessor extension of CARER allows a processor to write into its private cache between checkpoints. Checkpointing, which flushes all modified blocks, is performed when ownership of a block modified since the last checkpoint changes. *SafetyNet* is more efficient, since it does not checkpoint before every ownership transfer.

Other hardware BER schemes rely entirely upon memory to hold recoverable checkpoint state. The Sequoia computer system [10] uses private caches to hold state between checkpoints. The memory holds the consistent (checkpoint) state, and all dirty cache blocks are flushed to the main memory at every checkpoint. ReVive [82] employs a similar scheme, although it can tolerate the loss of a node by distributing memory and its parity across the nodes. Banâtre et al. [7] describe a scheme that is identical to a normal bus-based SMP, except that the traditional memory module has been replaced by an RSM (Recoverable Shared Memory) module. RSM requires a shadow copy of the entire memory as well as a mechanism for maintaining the inter-processor dependence graph to establish consistent recovery points. *SafetyNet* differs from all of these schemes by allowing checkpoint state to reside in the caches and by not requiring cache flushes to memory at every checkpoint.

IEEE's Scalable Coherent Interface (SCI) standard specifies potential hardware support for backward error recovery [49], but this recovery is limited to localized SCI ringlets. The designers deemed hardware support for end-to-end transaction recovery to be likely to introduce more problems than it would solve.

6.1.2 Software Backward Error Recovery

Software checkpointing have been developed, at radically different engineering costs from hardware BER schemes. In this section, we discuss checkpointing in software distributed shared memory (DSM) systems and in more general contexts.

Software DSM, as the name suggests, is a software implementation of shared memory. Accordingly, there are software schemes that provide support for improving the availability of these systems. Sultan et al. [104] develop a fault tolerance scheme for a software DSM scheme with the home-based lazy release consistency (HLRC) memory model. Wu and Fuchs [121] use a twin-page disk storage system to perform user-transparent checkpoint/recovery. At any point in time, one of the two disk pages is the working copy and the other page is the checkpoint. Similarly, Kim and Vaidya [55] develop a scheme that ensures that there are at least two copies of a page in the system. Morin et al. [67] leverage a Cache Only Memory Architecture (COMA) to ensure that at least two copies of a block exist at all times; traditional COMA

schemes ensure the existence of only one copy. Feeley et al. [33] implement log-based coherence for a transactional DSM.

Software checkpointing has also been developed for systems that do not employ DSM. Tandem machines prior to the S2 (e.g., the Tandem NonStop) use a checkpointing scheme in which every process periodically checkpoints its state on another processor [92]. If a processor fails, its processes are restarted on the other processors that hold the checkpoints. Condor [63], a batch job management tool, can checkpoint jobs in order to restart them on other machines. Applications need to be linked with the Condor libraries so that Condor can checkpoint them and restart them. Other schemes, including work by Plank [78, 79] and Wang and Hwang [112, 111], use software to periodically checkpoint applications for purposes of fault tolerance. These schemes differ from each other primarily in the degree of support required from the programmer, linked libraries, and the operating system.

IEEE's Scalable Coherent Interface (SCI) standard specifies software support for backward error recovery [49]. SCI can perform end-to-end error retry on coherent memory transactions, although the specification describes error recovery as being "relatively inefficient." Recovery is further complicated for SCI accesses to its non-coherent control and status registers (CSRs), since some of these actions may have side-effects.

SafetyNet differs from all of these works in that it is a hardware solution with different engineering costs/benefits. There exist similarities in that *SafetyNet* and these schemes both implement checkpoints, but software schemes can be much more elaborate. *SafetyNet* can be used as a complementary piece of an availability scheme that also includes software checkpoint/recovery. Integrating *SafetyNet* with a software scheme is an interesting area of future research, because it is likely to require at least some cooperation between the two levels of availability mechanisms.

6.1.3 Message Passing Backward Error Recovery

Numerous BER schemes exist for message passing systems, and they can be classified based on whether checkpointing is coordinated/consistent or not. Elnozahy et al. [31] provide an excellent tutorial and survey of this area of research, which we will now discuss in some more detail.

In uncoordinated/independent checkpointing schemes, such as Manetho [32], processors independently decide when to take checkpoints. There is no overhead for coordinating checkpoints, but these schemes are susceptible to rollback propagation (i.e., cascading rollbacks). Uncoordinated schemes can log incoming messages with approaches that are either pessimistic or optimistic. Pessimistic logging involves synchronously logging every message before processing it. Logging is thus costly, but the recovery scheme is much simpler and output commit is much faster. Optimistic logging assumes failures are rare, so it logs

incoming messages asynchronously. The logging cost is less, but the recovery scheme is complicated and output commit requires coordination.

In coordinated/consistent checkpointing schemes, processors must agree when to take a global checkpoint. This is more similar to *SafetyNet*, although *SafetyNet* implicitly coordinates in logical time, whereas these schemes coordinate in physical time. Koo and Toueg's scheme [56] uses an exchange of messages to coordinate checkpointing, whereas several other schemes assume synchronized physical clocks to coordinate checkpointing without an exchange of messages [84, 23].

6.1.4 (Hardware) Forward Error Recovery

FER schemes use redundant hardware to mask errors. ECC is the canonical FER scheme, and it uses redundant bits to mask bit errors. A typical FER scheme is triple modular redundancy (TMR), in which three identical components feed their results into a majority voter. Thus, TMR can mask a single error (except in the voter itself). Other FER schemes can be used to detect errors (requiring only duplicate redundancy) or mask more than just a single error (with higher degrees of redundancy). Most heavyweight fault tolerance schemes employ redundancy, and some lightweight schemes employ lighter redundancy (e.g., redundant threads instead of redundant processors).

At the processor level, numerous FER schemes exist for detecting errors and tolerating the faults that cause them. Redundant processors [6, 53, 54, 117] or redundant threads within a processor [106, 110] can be used to detect and/or mask processor faults. The Stratus [117] computer system uses two pairs of processors to mask errors. Within each pair, the two processors compare results—if the results do not match, an error has been detected and the other pair is now responsible. The Tandem S2 [53] uses triply modular redundant (TMR) processors to mask errors. Slipstream [106] is a lighter-weight processor FER scheme that can use redundant threads within a processor to mask errors. DIVA [6] uses a checker processor to implement FER on the processor (but not on the system).

FER schemes can also be used beyond just the processor. The Intel 432 [54] uses replication of VLSI components (i.e., commodity parts) to achieve a range of fault tolerance needs across the system. Interconnection networks have long used redundant paths and adaptive routing to allow packets to be routed around faulty switches and links [26, 30]. At the disk level and more recently at the DRAM level, RAID (redundant array of inexpensive disks [73] or DRAMs [28]) has been used to mask errors. RAID schemes have various flavors, known as levels, which trade off redundancy costs for fault tolerance capabilities.

6.2 Designability

Designability has not been explored in great depth, although a couple of recent papers have addressed it in the context of dynamic verification. DIVA [6], discussed in Section 6.1.4 for its use in availability, also addresses designability. The simple checker processor ensures that the system will function correctly even if the highly optimized core processor has a design fault. Other recent research seeks to dynamically verify complex cache coherence protocols by implementing checker coherence controllers that simultaneously run a much simpler version of the optimized protocol [16]. This research strives to extend designability support beyond the processor core.

6.3 Checkpoint/Recovery and Versioning of Data for Other Purposes

While checkpoint/recovery is clearly useful for supporting availability and designability, it serves purposes in numerous other areas of computer architecture and, more generally, computer science. In architecture, prior research for supporting speculation has logged changes in state that is local to a given node. Dynamically scheduled processors, such as the MIPS R10000 [122], must either log changes to architectural state or checkpoint architectural state, in case they need to recover from mis-speculations. Since dynamically scheduled processors are limited in terms of how much speculative state they can hold (i.e., in their reorder buffers), other research has sought to extend the amount of speculative state that can be maintained, so as to enable deeper speculation. SC++ [39] augments the reorder buffer with a Speculative History Queue that logs changes to the cache state due to speculative stores. In the case of a misspeculation, the actions logged in the Speculative History Queue are undone. Speculative Retirement [85] uses a speculative history buffer to speculatively retire instructions that would otherwise clog up the reorder buffer. Unlike SC++ that logs only store instructions, Speculative Retirement logs every speculative instruction that writes to a register. As with dynamically scheduled processors, both SC++ and Speculative Retirement require a mechanism to detect if another processor's store would violate the local speculation. *SafetyNet*'s logging is logically similar to these schemes, except *SafetyNet* is a *global* scheme; thus, *SafetyNet* must locally log transfers of coherence ownership and globally coordinate checkpoints across nodes. Implementations of *SafetyNet* can also leverage some properties of the application (availability/designability) that differ from speculation. For example, *SafetyNet* can checkpoint at a coarse granularity and not optimize the recovery process, since recovery due to errors is presumably far less frequent than violations due to misspeculation.

Beyond traditional (uniprocessor) speculation, the area of *speculative multithreading* uses data versioning (often implemented with logging) to implement sequential program semantics [3, 19, 41, 70, 81, 103]. In speculative multithreading, different processors (or processing elements within a processor) are assigned different sequential tasks which they speculatively execute in parallel. Thus, the same address can have

multiple outstanding values at the different processors, and some versioning control is necessary to determine the correct order of the values and to detect violations of sequential ordering due to misspeculation. The goal of *SafetyNet* differs in that we superimpose checkpoints on system execution with *parallel semantics*. We use globally consistent checkpoints rather than local checkpoints at different places in a sequential execution.

Beyond architecture, the concepts of checkpointing and logging have been used in various contexts. Most notably, databases use (software) checkpoint/recovery to ensure that data is never lost or corrupted [83], and they can use data versioning to maintain serializability [71].

6.4 Using Logical Time to Coordinate Multiprocessor Systems

The use of logical time to coordinate events in a multiprocessor system originated with Lamport's seminal paper [58]. This paper first described how to construct a logical time base to order events in a message passing system, and the fundamental idea is to use an algorithm that assigns a greater time to Event A than Event B if Event A is causally after Event B. For example, if processor 1 sends a message to processor 2, then the reception of the message (Event A) should occur after the sending of the message (Event B).

Systems have been designed that exploit logical time. Isotach networks [88] provide complete control over the logical ordering of messages in the network. Messages arrive at the (logical) time of the sender plus the (logical) distance to the receiver. Logical ordering is achieved by conservatively stalling messages (in physical time) in the network so that they arrive at the correct logical time at the destination. Logical ordering of messages enables totally ordered multicasts/broadcasts and the ability to make a group of operations (called an *isochron*) atomic in logical time. Delta cache coherence protocols [116, 27] exploit the strong ordering of Isotach networks to provide SC and powerful synchronization primitives.

Other research that we have done has leveraged logical time to devise new coherence protocols. Timestamp snooping [65] enables the use of broadcast snooping cache coherence protocols on systems with interconnection networks that do not support totally-ordered broadcasts. Instead of relying on a total order in physical time, timestamp snooping creates a total order in logical time. Logical time can be implemented in a variety of ways, including token-passing schemes. Optimizations of timestamp snooping allow nodes to process incoming requests early (i.e., before they arrive in logical time). We have developed other snooping cache coherence protocols that are derived from timestamp snooping and do not require a physical total order of coherence requests, although this feature is not emphasized in the original papers. Multicast snooping [11, 101] is a variant of timestamp snooping (as well as broadcast snooping) that leverages the logical total ordering of messages to allow processors to independently determine if a multicasted

request succeeds (i.e., is sent to all destinations that need to observe the request). Bandwidth adaptive snooping [66] is a variant of multicast snooping in which requests are either unicast or broadcast based on estimations of dynamic interconnection network utilization.

Logical time has also been applied to parallel discrete event simulation (PDES), in order to simulate a multiprocessor target system on a multiprocessor host system [35]. To manage the discrepancy between physical and logical times, PDES must determine, for each node, whether that node can process the "next" event on its event queue, because it may not know yet if another node will generate an event for it that should occur earlier (in logical time) than any event currently in its queue. Conservative schemes, such as the Wisconsin Wind Tunnel [86], nodes exchange information to determine the *global logical time*, so that processors can ensure that they only process the next event in logical time. The WWT breaks up target system simulation into quanta whose length, *Q*, is less than the minimum latency of the target's interconnection network. Keeping the *lookahead* (i.e., the minimum target time between events and the events that they can generate in remote nodes) greater than *Q* ensures that, at the beginning of each quantum, a host node has received all remotely generated events that could affect the target node in that quantum. Optimistic schemes, such as Chandrasekaran and Hill's extension of the WWT [17], let processors speculatively process events before determining the global logical time. If it later turns out that events were processed out of order, the system recovers to a previous state. Optimistic schemes can thus outperform conservative schemes if the cost of recoveries is less than the cost of waiting for global logical time to advance. This tradeoff is the same FER versus BER tradeoff that was described in Section 4.2 and illustrated in Figure 4-1.

Chapter 7

Summary

While architectural research has generally focused on improving performance, the issues of availability and designability have suffered. For both technological and architectural reasons, computer systems are more susceptible to hardware device faults. Meanwhile, systems are becoming increasingly difficult to design and verify as they become more complex in their efforts to achieve greater performance.

In this thesis, we develop a scheme, called *SafetyNet*, that unifies the support for improving the availability and designability of shared memory multiprocessors. *SafetyNet* is a system-wide, hardware-only, checkpoint/recovery scheme that enables a shared memory multiprocessor to recover to a pre-error checkpoint when an error is detected. Periodically, *SafetyNet* logically checkpoints the state of the system. The recovery point checkpoint, which is the checkpoint that was most recently validated as error-free, is the checkpoint to which the system recovers in the case that an error is detected. In between the recovery point and the active checkpoint, there are some number of old checkpoints that are pending validation.

In developing *SafetyNet*, this thesis makes three contributions which allow *SafetyNet* to be efficient in the common case of error-free execution. First, as opposed to previous hardware schemes for backward error recovery, *SafetyNet* uses logical time to efficiently coordinate creation of consistent checkpoints across the system. Second, *SafetyNet* uses a form of optimized logging to minimize the saving of checkpoint state. Third, *SafetyNet* enables the system to validate checkpoints in the background of the active execution, thus hiding the potentially lengthy error detection latency.

We describe an implementation of *SafetyNet*, and we address the implementation issues involved with *SafetyNet*. The implementation described in this thesis is based on a MOSI directory-based cache coherence protocol, with nodes connected by a two-dimensional torus interconnection network. We add a checkpoint log buffer (CLB) to each cache hierarchy and each memory, for purposes of logging changes to the memory and coherence state. Other additions are made to protect the system with *SafetyNet*, but these changes are minor.

We evaluate *SafetyNet* with full-system simulation and commercial workloads. We demonstrate that *SafetyNet* incurs negligible performance overhead, relative to an unprotected system, because of the innovations that allow it to be efficient. We show that 512 kbyte CLBs are sufficient, for our workloads and

100,000 cycle checkpoint intervals, to avoid a significant amount of stalling due to filling the CLBs. We also perform several sensitivity analyses to explore *SafetyNet*'s behavior for different implementation parameters. Notably, we evaluate the effects of changing the checkpoint interval length, register checkpointing latency, and CLB sizing.

We discuss how *SafetyNet* improves availability for a variety of error models. We first explore the interaction of *SafetyNet* with traditional error detection mechanisms. We discuss specific error models in the system—including the interconnection network, cache coherence protocol, and the processors—and how to detect these errors. Then we innovate in the area of error detection, by leveraging *SafetyNet*'s ability to tolerate error detection latencies on the order of hundreds of thousands of cycles. Given this latency tolerance, error detection can be extended to incorporate global mechanisms. For example, we develop two system-wide signature analysis schemes that perform global reductions to verify that the system obeys certain invariants. The message-level scheme detects if a message is lost or re-ordered in a broadcast snooping system, and the coherence-level scheme detects if a coherence upgrade is not matched by appropriate coherence downgrades. Both the message-level and coherence-level signature analysis schemes help to demonstrate the power of end-to-end invariant checking.

We also discuss how *SafetyNet* can improve system designability. We first classify the types of errors due to design faults that we address, dividing them broadly into errors due to unintentional design faults and errors due to speculatively correct design. For example, we speculatively design an adaptively routing interconnection network for a system whose cache coherence protocol requires point-to-point ordering. The adaptive routing can lead to violations of point-to-point ordering, but we use *SafetyNet* to recover from the rare situations in which reordering occurs and this reordering affects correctness. Meanwhile, we have enabled the use of adaptive routing, which can improve system performance by routing messages around congestion so as to better balance the traffic load in the interconnection network.

Future work exists in both availability and designability, since this thesis has not exhausted either of these areas. The availability research presented here does not address certain harder error models, most notably the permanent loss of a processor/cache chip. Relaxing some assumptions about what state is guaranteed safe opens up new areas of research. Also, availability research can be pursued in novel error detection schemes, moving beyond the signature analysis schemes presented in this thesis. Designability is an even more open research area. Work is still to be done in tolerating unintentional design faults that slip past verification and testing. In terms of speculatively correct designs, there are also numerous avenues of future research.

References

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, 1990.

[2] Rana E. Ahmed, Robert C. Frazier, and Peter N. Marinos. Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing Systems*, pages 82–88, June 1990.

[3] Haitham Akkary and Michael A. Driscoll. A Dynamic Multithreading Processor. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–236, November 1998.

[4] Alaa R. Alameldeen, Carl J. Mauer, Min Xu, Pacia J. Harper, Milo M.K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, February 2002.

[5] R. Anglada and A. Rubio. An Approach to Crosstalk Effect Analyses and Avoidance Techniques in Digital CMOS VLSI Circuits. *International Journal of Electronics*, 6(5):9–17, 1988.

[6] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, November 1999.

[7] M. Banâtre, A. Gefflaut, P. Joubert, P. Lee, and C. Morin. An Architecture for Tolerating Processor Failures in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 45(10):1101–1115, October 1996.

[8] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[9] Robert M. Bentley. Validating the Pentium 4 Microprocessor. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 493–498, July 2001.

[10] P. Bernstein. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, 21(2):37–45, February 1988.

[11] E. Ender Bilir, Ross M. Dickson, Ying Hu, Manoj Plakal, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Net-

work. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 294–304, May 1999.

[12] Manuel Blum and Sampath Kannan. Designing Programs that Check Their Work. In *ACM Symposium on Theory of Computing*, pages 86–97, May 1989.

[13] Manuel Blum and Hal Wasserman. Reflections on the Pentium Bug. *IEEE Transactions on Computers*, 45(4):385–393, April 1996.

[14] M. Bohr. Interconnect Scaling - The Real Limiter to High Performance. In *Proceedings of the International Electron Devices Meeting*, pages 241–244, December 1995.

[15] Philip Buonadonna and David Culler. Queue Pair IP: A Hybrid Architecture for System Area Networks. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 247–256, May 2002.

[16] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Dynamic Verification of Cache Coherence Protocols. In *Workshop on Memory Performance Issues*, June 2001. In conjunction with ISCA.

[17] Sashikanth Chandrasekaran and Mark D. Hill. Optimistic Simulation of Parallel Architectures Using Program Executables. In *Proceedings of Tenth Workshop on Parallel and Distributed Simulation (PADS '96)*, pages 143–150, May 1996.

[18] Alan Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 18(1):39–49, Jan/Feb 1998.

[19] Marcelo Cintra, Jose Martinez, and Josep Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

[20] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[21] B. Colwell. Maintaining a Leading Position. *IEEE Computer*, pages 45–47, January 1998.

[22] Anne E. Condon, Mark D. Hill, Manoj Plakal, and Daniel J. Sorin. Using Lamport Clocks to Reason About Relaxed Memory Models. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 270–278, January 1999.

[23] F. Cristian and F. Jahanian. A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations. In *Proceedings of IEEE Symposium on Reliable Distributed Systems*, pages 12–20, 1991.

[24] David E. Culler and J.P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[25] William J. Dally. Virtual Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[26] William J. Dally, Larry R. Dennison, David Harris, Kinhong Kan, and Thucydides Xanthopoulos. Architecture and Implementation of the Reliable Router. In *Proceedings of 2nd Hot Interconnects Symposium*, August 1994.

[27] Bronis R. de Supinski. *Logical Time Coherence Maintenance*. PhD thesis, University of Virginia, May 1998.

[28] Timothy J. Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. IBM Microelectronics Division Whitepaper, November 1997.

[29] Jose Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, December 1993.

[30] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks*. IEEE Computer Society Press, 1997.

[31] E.N. Elnozahy, D.B. Johnson, and Y.M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, September 1996.

[32] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[33] M.J. Feeley, J.S. Chase, V.R. Narasayya, and H.M. Levy. Integrating Coherency and Recoverability in Distributed Systems. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 215–227, November 1994.

[34] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Memory-access Times. *Electronics*, 57(1):164–169, January 1984.

[35] Richard M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[36] Mike Galles. Spider: A High-Speed Network Interconnect. *IEEE Micro*, 17(1):34–39, Jan/Feb 1997.

[37] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Von Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.

[38] Phillip B. Gibbons and Ephraim Korach. Testing Shared Memories. *SIAM Journal on Computing*, 26(4):1208–1244, August 1997.

[39] Chris Gniady, Babak Falsafi, and T.N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[40] S. W. Golumb. *Shift Register Sequences*. Aegean Park Press, revised edition, 1982.

[41] Sridhar Gopal, T.N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[42] G. Grohoski. Reining in Complexity. *IEEE Computer*, pages 41–42, January 1998.

[43] Rajiv Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, April 1989.

[44] R.N. Gustafson and F.J. Sparacio. IBM 3081 Processor Unit: Design Considerations and Design Process. *IBM Journal of Research and Development*, 26:12–21, January 1982.

[45] Erik Hagersten and Michael Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, January 1999.

[46] Robert H. Havemann and James A. Hutchby. High-Performance Interconnects: An Integration Overview. *Proceedings of the IEEE*, 89(5):586–601, May 2001.

[47] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[48] D.B. Hunt and P.N. Marinos. A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems*, pages 170–175, 1987.

[49] IEEE Computer Society. *IEEE Standard for Scalable Coherent Interface (SCI)*, August 1993.

[50] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual*, January 1996.

[51] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual, Volume 2: IA-64 System Architecture, Revision 1.1*, July 2000.

[52] iROC Technologies. White Paper on VDSM IC Logic and Memory Signal Integrity and Soft Errors, January 2002.

[53] D. Jewett. Integrity S2: A Fault-Tolerant UNIX Platform. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing Systems*, pages 512–519, June 1991.

[54] D. Johnson. The Intel 432: A VLSI Architecture for Fault-Tolerant Computing. *IEEE Computer*, pages 40–48, August 1984.

[55] J.-H. Kim and N.H. Vaidya. Recoverable Distributed Shared Memory Using the Competitive Update Protocol. In *Pacific Rim International Symposium on Fault-Tolerant Systems*, December 1995.

[56] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[57] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, pages 213–226, June 1981.

[58] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[59] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[60] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[61] David D. Lee and Randy H. Katz. Using Cache Mechanisms to Exploit Nonrefreshing DRAM's for On-Chip Memories. *IEEE Journal of Solid-State Circuits*, 26(4):657–66, April 1991.

[62] Charles E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[63] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, Computer Sciences Department, University of Wisconsin–Madison, April 1997.

[64] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[65] Milo M. K. Martin, Daniel J. Sorin, Anastassia Ailamaki, Alaa Alameldeen, Ross M. Dickson, Carl J. Mauer, Kevin E. Moore, Manoj Plakal, Mark D. Hill, and David A. Wood. Timestamp Snooping: An Approach for Extending SMPs. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, November 2000.

[66] Milo M. K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Bandwidth Adaptive Snooping. In *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, pages 251–262, January 2002.

[67] C. Morin, A. Gefflaut, M. Banatre, and A.-M. Kermarrec. COMA: An Opportunity for Building Fault-Tolerant Scalable Shared Memory Multiprocessors. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pages 56–65, May 1996.

[68] Shubhendu S. Mukherjee, Peter Bannon, Steven Lang, Aaron Spink, and David Webb. The Alpha 21364 Network Architecture. In *Proceedings of 9th Hot Interconnects Symposium*, August 2001.

[69] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed Design and Implementation of Redundant Multhreading Alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 99–110, May 2002.

[70] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor. Technical Report CSL-TR-97-715, Stanford University, May 1997.

[71] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Maryland, 1986.

[72] David A. Patterson. Recovery Oriented Computing: A New Research Agenda for a New Century. HPCA-8 Keynote Address, January 2002.

[73] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of 1988 ACM SIGMOD Conference*, pages 109–116, June 1988.

[74] Fernando Pedone. Boosting System Performance with Optimistic Distributed Protocols. *IEEE Computer*, pages 80–86, December 2001.

[75] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 1996.

[76] W. W. Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. MIT Press, 1972.

[77] Manoj Plakal, Daniel J. Sorin, Anne E. Condon, and Mark D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proceedings of the Tenth ACM Symposium on Parallel Algorithms and Architectures*, pages 67–76, June 1998.

[78] James S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, July 1997.

[79] James S. Plank, Kai Li, and Michael A. Puening. Diskless Checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[80] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, Inc., 1996.

[81] Milos Prvulovic, Maria Jesus Garzaran, Lawrence Rauchwerger, and Josep Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 204–215, July 2001.

[82] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, May 2002.

[83] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems, 2nd edition.* McGraw-Hill, 1999.

[84] P. Ramanathan and K.G. Shin. Checkpointing and Rollback Recovery in a Distributed System Using Common Time Base. In *Proceedings of the 7th Symposium on Reliable Distributed Systems*, pages 13–21, October 1988.

[85] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210, June 1997.

[86] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[87] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.

[88] Paul F. Reynolds, Jr., Craig Williams, and Raymond R. Wagner, Jr. Isotach Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):337–348, April 1997.

[89] Jack Robertson. Alpha Particles Worry IC Makers as Device Features Keep Shrinking. *Semiconductor Business News*, October 21, 1998.

[90] Jack Robertson. Sun Confirms Glitch in UltraSparcIII Processor. *Silicon Strategies*, April 4, 2001.

[91] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems*, pages 84–91, June 1999.

[92] O. Serlin. Fault-Tolerant Systems in Commercial Applications. *IEEE Computer*, pages 19–30, August 1984.

[93] K. Seshan, T. Maloney, and K. Wu. The Quality and Reliability of Intel's Quarter Micron Process. *Intel Technology Journal*, September 1998.

[94] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002.

[95] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[96] Timothy J. Slegel et al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, pages 12–23, March/April 1999.

[97] James E. Smith and Andrew R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, C-37(5):562–573, May 1988.

[98] Gurindar S. Sohi, Manoj Franklin, and Kewal K. Saluja. A Study of Time-Redundant Fault Tolerance Techniques for High-Performance Pipelined Computers. In *Proceedings of the 19th International Symposium on Fault-Tolerant Computing Systems*, pages 436–443, June 1989.

[99] Daniel J. Sorin, Milo M.K. Martin, Mark D. Hill, and David A. Wood. Fast Checkpoint/Recovery to Support Kilo-Instruction Speculation and Hardware Fault Tolerance. Technical Report 1420, Computer Sciences Department, University of Wisconsin–Madison, October 2000.

[100] Daniel J. Sorin, Milo M.K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, May 2002.

[101] Daniel J. Sorin, Manoj Plakal, Mark D. Hill, Anne E. Condon, Milo M.K. Martin, and David A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, June 2002.

[102] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(5/6), September/November 1999.

[103] J. Gregory Steffan and Todd C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[104] Florin Sultan, Thu Nguyen, and Liviu Iftode. Scalable Fault-Tolerant Distributed Shared Memory. In *Proceedings of SC2000*, November 2000.

[105] Sun Microsystems. *UltraSPARC User's Manual*. Sun Microsystems, Inc., July 1997.

[106] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, November 2000.

[107] Y. Tamir and M. Tremblay. High-Performance Fault-Tolerant VLSI Systems Using Micro Rollback. *IEEE Transactions on Computers*, 39(4):548–554, April 1990.

[108] Scott Taylor et al. Functional Verification of a Multiple-Issue, Out-of-Order, Superscalar Alpha Processor–The DEC Alpha 21264 Microprocessor. In *Design Automation Conferehce*, pages 638–643, June 1998.

[109] M. Tremblay. Increasing Work, Pushing the Clock. *IEEE Computer*, pages 40–41, January 1998.

[110] T. N. Vijaykumar, Irith Pomeranz, and Karl K. Chung. Transient Fault Recovery Using Simultaneous Multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 87–98, May 2002.

[111] Y. M. Wang, E. Chung, Y. Huang, and E.N. Elnozahy. Integrating Checkpointing with Transaction Processing. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing Systems*, pages 304–308, June 1997.

[112] Y-M. Wang, Y. Huang, K-P. Vo, P-Y. Chung, and C. Kintala. Checkpointing and Its Applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems*, pages 22–31, June 1995.

[113] David L. Weaver and Tom Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[114] N. Weste and K. Eshragian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley Publishing Co., 1982.

[115] George White and Pete Vogt. Profusion: A Buffered, Cache Coherent Crossbar Switch. In *Proceedings of 5th Hot Interconnects Symposium*, pages 87–96, August 1997.

[116] Craig Williams. *Concurrency Control in Asynchronous Computations*. PhD thesis, University of Virginia, Computer Sciences Department, January 1993.

[117] D. Wilson. The Stratus Computer System. In *Resilient Computer Systems*, pages 208–231, 1985.

[118] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[119] David A. Wood, Garth A. Gibson, and Randy H. Katz. Verifying a Multiprocessor Cache Controller Using Random Test Generation. *IEEE Design and Test of Computers*, pages 13–25, August 1990.

[120] K. Wu, W. K. Fuchs, and J. H. Patel. Error Recovery in Shared Memory Multiprocessors Using Private Caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):231–240, April 1990.

[121] K.-L. Wu and W. K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[122] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[123] J. Ziegler et al. IBM Experiments in Soft Fails in Computer Electronics. *IBM Journal of Research and Development*, 40(1):3–18, January 1996.

Appendix A

Tabular Specification of *SafetyNet* Directory Protocol

In this appendix, we fully specify the *SN-Directory* cache coherence protocol. The specification is in a tabular format that was developed by Sorin et al. [101], and an online version of this specification is available at `http://www.cs.wisc.edu/multifacet/public/sorin_thesis/`. For each controller—cache controller, directory/memory controller, and network interface controller—we specify four tables that describe the controller's behavior with respect to any given block:

- States:[1] States are specified as one or more letters. For example, the transient state OI denotes that a cache controller was in state Owned and then issued a Put-Exclusive (PUTX).
- Actions: Actions are specified as individual letters. For example, the letter *a* could denote that the cache controller allocates a transaction buffer entry (TBE).
- Events: Events are triggered by incoming messages. Messages arrive on queues, as shown for all controllers in Figure A-1. At the cache, incoming queues are the Mandatory queue (for requests from the processor), the Service processor queue (for handling requests to the service processor, which is emulated by processor 0), the Recovery queue (for undoing log entries from the CLB and from the TBE), and queues from the network interface. At the directory, incoming queues are the Recovery queue and queues from the network interface. At the network interface, incoming queues are from the interconnection network and from the cache and directory controllers.
- Transitions: A transition—an intersection of a state on a row and an event on a column—is specified as a sequence of actions and a next state (if the state changes) separated by a slash. For example, the transition *ab/S* denotes that actions *a* and *b* are performed and that the next state is *S*. Transitions that are shaded are impossible. If any action in a sequence cannot be performed, due to a resource constraint such as being unable to allocate a transaction buffer entry (TBE), then the transition is not performed and none of the actions are performed.

There are two issues in the cache specification that should be noted. First, there is a single cache controller that manages both L1 caches and the L2 cache. Other implementations could have separate controllers for

1. The network interface controller has only one "state," so we omit this table.

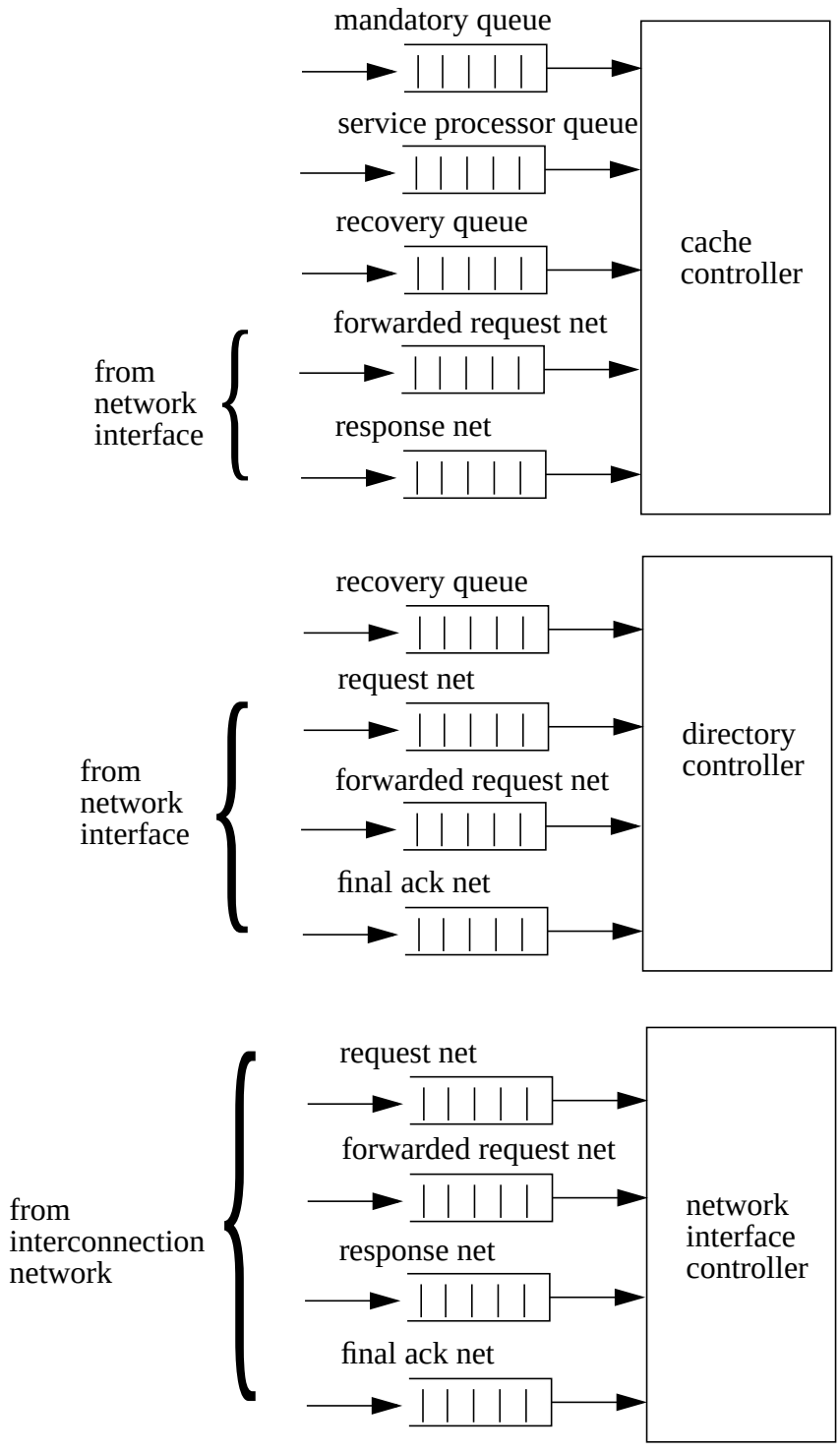

FIGURE A-1. Protocol state machines and their incoming queues

Table A-1. *SN-Directory* - cache controller states

State	Description
NP	Not present
I	Idle
S	Shared
O	Owned
M	Modified
MI	Modified, issued PUTX, have not seen response yet
OI	Owned, issued PUTX, have not seen response yet
IS	Idle, issued GETS, have not seen response yet
ISI	Idle, issued GETS, saw INV, have not seen data for GETS yet
IM	Idle, issued GETX, have not seen response yet
IM^n	Idle, issued GETX, saw nack, still waiting for acks
IMI	Idle, issued GETX, saw forwarded GETX
IMO	Idle, issued GETX, saw forwarded GETS
IMOI	Idle, issued GETX, saw forwarded GETS, saw forwarded GETX
OM	Owned, issued GETX, have not seen response yet
OM^n	OM, saw nack, still waiting for acks

each, but we opted for the simplicity of this design. Second, there are three transitions in the cache controller transition table that are shaded despite being specified. These transitions are, in general, impossible, but they can occur when we enable adaptive routing in the interconnection network (discussed in Section 5.1.2). As such, the actions for these transactions involve triggering a system recovery.

Table A-2. *SN-Directory* - cache controller actions

Action	Description	Action	Description
a	Issue GETS	ϕ	Record FwdGETX and ack count for forwarding
b	Issue GETX	κ	Restart system after recovery
c	Send FinalAck to dir if this is response to 3-hop xfer	λ	Copy block from head of Recovery queue to L2 cache
d	Issue PUTX	μ	Log upgrade of block in CLB
e	Send data from cache to requestor	ν	Record in TBE if Final-Ack will be needed
f	Issue GET_INSTR	o	Delayed precommit of a version
h	If not prefetch, notify sequencer the load completed.	π	Broadcast Pre-Restart message
i	Allocate TBE (isPrefetch=0, number of invalidates=0)	θ	Broadcast Restart message
j	Set prefetch bit	ρ	Log entry in CLB (from cache)
k	Pop mandatory queue.	σ	Log entry in CLB (from TBE)
l	Pop incoming forwarded request queue	τ	Update logical clock
m	Pop optional queue	υ	(profiling)
n	Send Final-Ack to directory if 3-hop transaction	ϖ	Broadcast Recovery message
o	Pop Incoming Response queue	ω	Pop incoming service processor queue
p	Add number of pending acks to TBE	ξ	Log entry in CLB (from TBE)
q	Decrement number of pending invalidations by one	ψ	Pop incoming Recovery queue
r	Recycle head of recovery queue (from CLB) to tail	ζ	Broadcast Timeout message
s	Deallocate TBE	A,B,C,D	(profiling)
t	Send ack to invalidator	E	Send Nack to requestor
u	Write data to cache	G	Reset the TBE
v	Check to see if space in CLB	H	Trigger system recovery
x	Copy data from cache to TBE	L	Copy data block from L2 to L1 (I or D)
y	Send data from TBE to requestor	M	Log CLB entry to record upgrade of block (from TBE)
z	Stall	Q	Send Final-Nack to directory if 3-hop transaction
α	Recover TBE that was used for PUTX transaction	R	Bookkeeping for multipass recovery from CLB
β	Commit a version	S	Set L1 D-cache tag equal to tag of block B.
χ	Recover system	T	Set L1 I-cache tag equal to tag of block B.
δ	Record forwarded GETS for future forwarding	U	Set L2 cache tag equal to tag of block B.
ε	Send data from cache to GetS ForwardIDs	V	Send Final-Nack to directory if 3-hop transaction
ϕ	Record forwarded GETX and ack count for future forwarding	W	Send Final_nack if Nack was from myself
γ	Send data from cache to GetX ForwardID	X	Deallocate L1 cache block. Sets the cache to not present, allowing a replacement in parallel with a fetch.
η	If not prefetch, notify sequencer that store completed.	Y	Deallocate L2 cache block. Sets the cache to not present, allowing a replacement in parallel with a fetch.
ι	Count a PreCommit for a checkpoint number	Z	Copy data block from L1 (I or D) to L2

Table A-3. *SN-Directory* - cache controller events

Event	Description
Load	Load request from the processor
Load_prefetch	Load prefetch request from the processor
Ifetch	I-fetch request from the processor
Store	Store request from the processor
Store_prefetch	Store prefetch request from the processor
L1_to_L2	L1 to L2 transfer
L2_to_L1D	L2 to L1-Data transfer
L2_to_L1I	L2 to L1-Instruction transfer
L2_Replacement	L2 Replacement
Forwarded GET_INSTR	Directory forwards GET_INSTR to us
Forwarded GETS	Directory forwards GETS to us
Forwarded GETX	Directory forwards GETX to us
INV	Invalidation
CLBstall	Cannot process Forwarded-GETX due to filling CLB
Proc ack	Ack from processor
Proc last ack	Last ack from a processor
Data ack 0	Data with ack count = 0
Data ack not 0	Data with ack count != 0 (but haven't seen all acks first)
Data ack not 0 last	Data with ack count != 0 after having received all acks
WB ack	Writeback ack from directory
Dir nack 0	Nack with ack count = 0
Dir nack not 0	Nack with ack count != 0 (but haven't seen all acks first)
Dir nack not 0 last	Nack with ack count != 0 after having received all acks
DelayedPreCommit	Cache just now became ready to PreCommit
Commit	Commit a version
Recovery	Recover system to recovery point checkpoint
PreRestart	Pre-Restart system after a recovery
Restart	Restart system after a recovery
Timeout	Timeout
RecoverStaleI_cacheAvail	Recover StaleI data into cache
RecoverStaleO_cacheAvail	Recover StaleO data into cache
RecoverStaleM_cacheAvail	Recover StaleM data into cache
RecoverStaleI_cacheNotAvail	Cache not available for StaleI data
RecoverStaleO_cacheNotAvail	Cache not available for StaleO data
RecoverStaleM_cacheNotAvail	Cache not available for StaleM data
UpdateRecyclingCount	After pass through CLB, increment counter
RecoverSpecTBE	Recover TBE that has uncommitted state
RecoverNonSpecTBE	Recover TBE that has committed state
ExtPreCommit	A PreCommit for a checkpoint number arrives at service processor
ReqTimeout	Service processor requests Timeout
ReqRecovery	Service processor requests Recovery
ReqPreRestart	Service processor requests Pre-Restart
ReqRestart	Service processor requests Restart
IgnoreCPU	Ignore request from mandatory queue during Recovery
IgnoreFwdReq	Ignore forwarded request msg during Recovery
IgnoreResponse	Ignore response msg during Recovery

Table A-8. *SN-Directory* - directory controller states

State	Description
NP	Not present
I	Idle
S	Shared
O	Owned
M	Modified
OO	Owned, saw GETS
OM	Owned, saw GETX
MO	Modified, saw GETS
MM	Modified, saw GETX

Table A-4. *SN-Directory* - cache controller transitions (part 1 of 4)

State	Load	Load_prefetch	Ifetch	Store	Store_prefetch	L1_to_L2	L2_to_L1D	L2_to_L1I	L2_Replacement	Forwarded GET_INSTR	Forwarded GETS
NP	τSiaʊk/ IS	τSijam/ IS	τTifʊk/ IS	τvSi-bʊk/IM	τvSijbm /IM	UZX	SLY	TLY	τY		
I	τSiaʊk/ IS	τSijam/ IS	τTifʊk/ IS	τvSi-bʊk/IM	τvSijbm /IM	UZX	SLY	TLY	τY		
S	τhk	τm	τhk	τvibʊk/ IM	τvijbm/ IM	UZX	SLY	TLY	τY/I		
O	τhk	τm	τhk	τvix-bʊk/OM	τvixjbm /OM	UZX	SLY	TLY	τvixdY/ OI	τAel	τAel
M	τhk	τm	τhk	τvηk	τm	UZX	SLY	TLY	τvixdY/ MI	τAel/O	τAel/O
MI	τz	τz	τz	τz	τz				τz	τAyl	τAyl
OI	τz	τz	τz	τz	τz				τz	τAyl	τAyl
IS	τz	τz	τz	τz	τz				τz		
ISI	τz	τz	τz	τz	τz				τz		
IM	τz	τz	τz	τz	τz				τz	τAδl/ IMO	τAδl/ IMO
IM^n	τz	τz	τz	τz	τz				τz		
IMI	τz	τz	τz	τz	τz				τz		
IMO	τz	τz	τz	τz	τz				τz	τAδl	τAδl
IMOI	τz	τz	τz	τz	τz				τz		
OM	τz	τz	τz	τz	τz				τz	τAel	τAel
OM^n	τz	τz	τz	τz	τz				τz	τAel	τAel

requests from processor

L1/L2 exchanges

forwarded requests

Table A-5. *SN-Directory* - cache controller transitions (part 2 of 4)

State	Forwarded GETX	INV	CLBstall	Proc ack	Proc last ack	Data ack 0	Data ack not 0	Data ack not 0 last	WB ack	Dir nack 0	Dir nack not 0	Dir nack not 0 last
NP	τ**AEHl**	τAtl										
I	τ**AEHl**	τAtl										
S		τAtl/I										
O	τBρel/I		τEl									
M	τBρel/I		τEl									
MI	τCξyl		τEl						τσsl/I	τdGo		
OI	τCξyl		τEl						τσsl/I	τdGo		
IS	τ**AEHl**	τAtl/ ISI				τuhsco/ S						
ISI		τAtl				τuhsco/I						
IM	τDϕl/ IMI	τAtl	τEl	τqo	τMηnso/M	τμuη–sco/M	τupνo	τμuη–sco/M		τQbGo	τpνo/ IM^n	τQbGo
IM^n		τAtl		τqo	τVbGo /IM							
IMI				τqo	τMη–γnso/I	τμuηγ–sco/I	τupνo	τμuηγ–sco/I				
IMO	τDϕl/ IMOI			τqo	τMηεnso/O	τμu-ηεsco/O	τupνo	τμu-ηεsco/O				
IMOI				τqo	τMηεγnso/I	τμuηεγ–sco/I	τupνo	τμuηεγ–sco/I				
OM	τBρel/ IM		τEl	τqo						τWbGo	τpνo/ OM^n	τWbGo
OM^n				τqo	τVbGo /OM							

forwarded requests (cont'd)

responses from cache

responses from directory

Table A-6. *SN-Directory* - cache controller transitions (part 3 of 4)

State	Commit	Recovery	PreRestart	Restart	Timeout	RecoverStaleI_cacheAvail	RecoverStaleO_cacheAvail	RecoverStaleM_cacheAvail	RecoverStaleI_cacheNotAvail	RecoverStaleO_cacheNotAvail	RecoverStaleM_cacheNotAvail	UpdateRecyclingCount
NP	τβl	τχl	τφl	τκl	τl	τψ/I	τλψ/O	τλψ/M	τrψ	τrψ	τrψ	R
I	τβl	τχl	τφl	τκl	τl	τψ	τλψ/O	τλψ/M	τrψ	τrψ	τrψ	R
S	τβl	τχl	τφl	τκl	τl	τψ/I						R
O	τβl	τχl	τφl	τκl	τl	τψ/I	τψ	τψ				R
M	τβl	τχl	τφl	τκl	τl	τψ/I	τψ	τψ				R
MI	τβl	τχl	τφl	τκl	τl		τψ	τψ				R
OI	τβl	τχl	τφl	τκl	τl		τψ	τψ				R
IS	τβl	τχl	τφl	τκl	τl							
ISI	τβl	τχl	τφl	τκl	τl							
IM	τβl	τχl	τφl	τκl	τl							
IM^n	τβl	τχl	τφl	τκl	τl							
IMI	τβl	τχl	τφl	τκl	τl							
IMO	τβl	τχl	τφl	τκl	τl							
IMOI	τβl	τχl	τφl	τκl	τl							
OM	τβl	τχl	τφl	τκl	τl							
OM^n	τβl	τχl	τφl	τκl	τl							

checkpoint management

recovering from CLB

Table A-7. *SN-Directory* - cache controller transitions (part 4 of 4)

State	RecoverSpecTBE	RecoverNonSpecTBE	ExtPreCommit	ReqTimeout	ReqRecovery	ReqPreRestart	ReqRestart	IgnoreCPU	IgnoreFwdReq	IgnoreResponse
NP	τsψ/I	τsψ/I	τιο	τζω	τϖω	τπω	τθω	k	l	o
I	τsψ	τsψ	τιο	τζω	τϖω	τπω	τθω	k	l	o
S	τsψ/I	τsψ/I	τιο	τζω	τϖω	τπω	τθω	k	l	o
O	τsψ	τsψ	τιο	τζω	τϖω	τπω	τθω	k	l	o
M	τsψ	τsψ	τιο	τζω	τϖω	τπω	τθω	k	l	o
MI	τsψ/I	ταψ	τιο	τζω	τϖω	τπω	τθω	k	l	o
OI	τsψ/I	ταψ	τιο	τζω	τϖω	τπω	τθω	k	l	o
IS	τsψ/I	τsψ/I	τιο	τζω	τϖω	τπω	τθω	k	l	o
ISI	τsψ/I	τsψ/I	τιο	τζω	τϖω	τπω	τθω	k	l	o
IM	τsψ/I	τsψ/I	τιο	τζω	τϖω	τπω	τθω	k	l	o
IM^n	τsψ/I	τsψ/I	τιο	τζω	τϖω	τπω	τθω	k	l	o
IMI	τsψ/I	τsψ/I	τιο	τζω	τϖω	τπω	τθω	k	l	o
IMO	τsψ/I	τsψ/I	τιο	τζω	τϖω	τπω	τθω	k	l	o
IMOI	τsψ/I	τsψ/I	τιο	τζω	τϖω	τπω	τθω	k	l	o
OM	τsψ/I	τsψ/O	τιο	τζω	τϖω	τπω	τθω	k	l	o
OM^n	τsψ/I	τsψ/O	τιο	τζω	τϖω	τπω	τθω	k	l	o

recovering from TBE

service processor requests

draining network

Table A-9. *SN-Directory* - directory controller actions

Action	Description
a	Add requestor to list of sharers
b	Send data to requestor
d	Forward request to owner
e	Deallocate TBE
f	Set owner equal to requestor
g	Clear list of sharers
h	Send Invalidations to all sharers
i	Allocate TBE
j	Pop incoming request queue
k	Pop incoming forwarded request queue
l	Write incoming data to memory
m	Nack incoming request
n	Send PUTX-Ack to requestor
p	Clear owner
r	Add owner to list of sharers
t	Remove owner from list of sharers
u	Remove requestor from list of sharers
v	Add CLB entry, if necessary
w	Add CLB entry, if necessary
x	Recycle request from head of incoming queue to tail
z	Stall
β	Recover directory to checkpointed state
χ	Restart system
δ	Commit a version
ε	Pre-Restart the System
η	Pop incoming final-ack queue
ι	Pop incoming recovery queue
φ	Deallocate TBE
κ	Recover state from TBE
o	Delayed precommit of a version
π	Final-Nack undoes 3-hop transaction
τ	Update logical clock
A	(profiling)
B	(profiling)
C	Trigger system recovery

Table A-10. *SN-Directory* - directory controller events

Event	Description
GETS	A GETS arrives
GET_INSTR	A GET_INSTR arrives
GETX_Owner	A GETX arrives, requestor is owner
GETX_NotOwner	A GETX arrives, requestor is not owner
PUTX (requestor is owner)	A PUTX arrives, requestor is owner
PUTX (requestor not owner)	A PUTX arrives, requestor is not owner
CLBstall	Stall due to full CLB
FinalAck	Final-Ack
FinalNack	Final-Nack
Commit	Commit a version
DelayedPreCommit	Delayed PreCommit
Recovery	System recovery
PreRestart	Pre-Restart system (phase 1 of recovery)
Restart	Restart system after a recovery (phase 2 of recovery)
Timeout	Timeout to advance logical time
RecoverTBE	Recover state from TBE
IgnoreRequestMsg	Ignore message during recovery
IgnoreFwdRequestMsg	Ignore message during recovery
IgnoreFinalAckMsg	Ignore message during recovery

Table A-11. *SN-Directory* - directory controller transitions

State	GETS	GET_INSTR	GETX_Owner	GETX_NotOwner	PUTX (requestor is owner)	PUTX (requestor not owner)	CLBstall	FinalAck	FinalNack	Commit	DelayedPreCommit	Recovery	PreRestart	Restart	Timeout	RecoverTBE	IgnoreRequestMsg	IgnoreFwdRequestMsg	IgnoreFinalAckMsg
NP	τAabj/S	τAabj/S		τBvfbj/M		τAnj	τmj			τδk	το	τβk	τεk	τχk	τk	τφι	j	k	η
I	τAabj/S	τAabj/S		τBvfbj/M		τAnj	τmj			τδk	το	τβk	τεk	τχk	τk	τφι	j	k	η
S	τAabj	τAabj		τBvubfhgj/M		τAnj	τmj			τδk	το	τβk	τεk	τχk	τk		j	k	η
O	τAi-adj/OO	τAi-adj/OO	τBiutdf-hgj/OM	τBiutdf-hgj/OM	τBvulnpj/S	τAnj	τmj			τδk	το	τβk	τεk	τχk	τk	τφι	j	k	η
M	τAiardj/MO	τAiardj/MO	τBZZj	τBidfj/MM	τBv-lnpj/I	τAnj	τmj			τδk	το	τβk	τεk	τχk	τk		j	k	η
OO	τxj	τxj	τxj	τxj	τxj	τxj	τmj	τεη/O	τπεη/O	τδk	το	τβk	τεk	τχk	τk	τκφι/O	j	k	η
OM	τxj	τxj	τxj	τxj	τxj	τxj	τmj	τweη/M	τπεη/O	τδk	το	τβk	τεk	τχk	τk	τκφι/O	j	k	η
MO	τxj	τxj	τxj	τxj	τxj	τxj	τmj	τεη/O	τπεη/M	τδk	το	τβk	τεk	τχk	τk	τκφι/O	j	k	η
MM	τxj	τxj	τxj	τxj	τxj	τxj	τmj	τweη/M	τπεη/M	τδk	το	τβk	τεk	τχk	τk	τκφgι/M	j	k	η

requests

final ack/nack

checkpoint management

draining network

Table A-12. *SN-Directory* - network interface actions

Action	Description
a	Send response message from cache to network
b	Send request message from cache to network
c	Send response message from dir to network
d	Send forwarded request message from dir to network
e	Send response message from network to cache or dir
f	Send request message from network to dir
g	Send forwarded request message from network to cache and dir
h	Pop Incoming Response Network
i	Pop Incoming Request Network
j	Pop Incoming Forwarded Request Network
k	Pop response queue from cache
l	Pop request queue from cache
m	Pop response queue from dir
n	Pop forwarded request queue from dir
o	Send forwarded request message from cache to network
p	Pop forwarded request queue from cache
q	Send Final-Ack from cache to network
r	Pop Final-Ack from cache queue
s	Send Final-Ack from network to directory
t	Pop incoming Final-Ack from network queue

Table A-13. *SN-Directory* - network interface events

Event	Description
OutgoingRequestFromCache	Outgoing cache request
OutgoingForwardedRequestFromCache	Outgoing cache forwarded request
OutgoingResponseFromCache	Outgoing cache response
OutgoingFinalAckFromCache	Outgoing cache final-ack
OutgoingForwardedRequestFromDir	Outgoing dir forwarded request
OutgoingResponseFromDir	Outgoing dir response
IncomingRequest	Incoming request
IncomingForwardedRequest	Incoming forwarded request
IncomingResponse	Incoming response
IncomingFinalAck	Incoming final-ack

Table A-14. *SN-Directory* - network interface transitions

State	OutgoingRequestFromCache	OutgoingForwardedRequestFromCache	OutgoingResponseFromCache	OutgoingFinalAckFromCache	OutgoingForwardedRequestFromDir	OutgoingResponseFromDir	IncomingRequest	IncomingForwardedRequest	IncomingResponse	IncomingFinalAck
I	bl	op	ak	qr	dn	cm	fi	gj	eh	st

from cache to network

from directory to network

from network to node