SCALABLE DISKLESS CHECKPOINTING FOR LARGE PARALLEL SYSTEMS

BY
CHARNG-DA LU

B.S., National Taiwan University, 1997
M.S., University of Illinois at Urbana-Champaign, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

SCALABLE DISKLESS CHECKPOINTING FOR LARGE PARALLEL SYSTEMS

Charng-da Lu
Department of Computer Science
University of Illinois at Urbana-Champaign, 2005
Daniel A. Reed, Advisor

Parallel scientific applications deal with machine unreliability by periodic checkpointing, in which all processes coordinate to dump memory to stable storage simultaneously. However, in systems comprising tens of thousands of nodes, the total data volume can overwhelm the network and storage farm, creating an I/O bottleneck. Furthermore, a very large class of scientific applications can fail on these systems if one of the processes dies.

Poor checkpointing performance limits checkpointing frequency and increases the time-to-solution of applications. Also, the application can spend more time in recovery and restart because large systems tend to fail often.

Diskless checkpointing is a viable approach that provides high-performance and reliable storage for *intermediate or temporary* data, such as checkpoint files. First, the data is stored in memory instead of disk. Second, reliability and recoverability is guaranteed by use of redundancy codes (parity bits or Reed-Solomon codes), which are stored on spares. Third, I/O is made scalable by partitioning nodes and spares into small groups. Each group takes care of its own redundancy codes generation and node failure and recovery.

We have implemented a diskless checkpointing and recovery system and assessed its performance with both I/O benchmarks and real scientific applications. The results show much greater I/O scalability and higher throughput than disk-based paralell file systems for a large number of clients.

As a technology projection, we have also developed an analytical model to investigate the performability of diskless checkpointing. Our model evaluation shows that the overhead of checkpoint/recovery is small on systems with thousands of nodes, and with appropriate partitioning of nodes, the user application can survive several times longer.

Acknowledgments

I wish to thank Professor Daniel A. Reed for his guidance and support in my research and development of this thesis. I would also like to thank Professor Marc Snir, Professor Ravishankar K. Iyer, and Pablo group members, Celso Mendes and Karthik Pattabiraman, for invaluable discussions and comments.

This research was supported partly by Contract 74837-001-0349 from the Regents of University of California (Los Alamos National Laboratory) to William Marsh Rice University and by National Science Foundation under grant ACI 02-19597.

Table of Contents

List of Tables

List of Figures

SCALABLE DISKLESS CHECKPOINTING FOR LARGE PARALLEL SYSTEMS

Charng-da Lu
Department of Computer Science
University of Illinois at Urbana-Champaign, 2005
Daniel A. Reed, Advisor

Parallel scientific applications deal with machine unreliability by periodic checkpointing, in which all processes coordinate to dump memory to stable storage simultaneously. However, in systems comprising tens of thousands of nodes, the total data volume can overwhelm the network and storage farm, creating an I/O bottleneck. Furthermore, a very large class of scientific applications can fail on these systems if one of the processes dies.

Poor checkpointing performance limits checkpointing frequency and increases the time-to-solution of applications. Also, the application can spend more time in recovery and restart because large systems tend to fail often.

Diskless checkpointing is a viable approach that provides high-performance and reliable storage for *intermediate or temporary* data, such as checkpoint files. First, the data is stored in memory instead of disk. Second, reliability and recoverability is guaranteed by use of redundancy codes (parity bits or Reed-Solomon codes), which are stored on spares. Third, I/O is made scalable by partitioning nodes and spares into small groups. Each group takes care of its own redundancy codes generation and node failure and recovery.

We have implemented a diskless checkpointing and recovery system and assessed its performance with both I/O benchmarks and real scientific applications. The results show much greater I/O scalability and higher throughput than disk-based paralell file systems for a large number of clients.

As a technology projection, we have also developed an analytical model to investigate the performability of diskless checkpointing. Our model evaluation shows that the overhead of checkpoint/recovery is small on systems with thousands of nodes, and with appropriate partitioning of nodes, the user application can survive several times longer.

Chapter 1

Introduction

1.1 Large System Reliability

One of the challenges in achieving petaflop (10^{15} floating-point operations per second) performance is system reliability. Although advances in VLSI technology have improved the reliability of commercial off-the-shelf (COTS) hardware, the availability of a COTS-based high-performance computing system is limited by a well-known fact in the reliability theory: the Mean Time to Failure (MTTF) shortens as component count escalates. When a system consists of tens of thousands of nodes, its MTTF can be as short as several hours.

As a motivating example, assume node failures follow exponential distributions and let R be a single node's one-hour reliability. Furthermore, suppose the system stops functioning if one node fails. In this scenario, an n-node system's MTTF is approximately $1/(1-R^n)$. Figure 1.1 plots the MTTF for different n's and R's.

Commodity components usually have one to three years of warranty. If we use this period as a node's MTTF, then R falls between 0.9999 and 0.99999. It is clear that when the system size approaches 10,000 nodes, the MTTF drops to less than 10 hours. Even for a system based on ultra-reliable components (MTTF of 114 years, or equivalently, $R = 0.999999$), the system's MTTF is less than 20 hours for a supercomputer like Blue Gene. [1]

[1] Announced in late 1999, the IBM Blue Gene project [1] plans to build a supercomputer of 65,536 nodes capable of

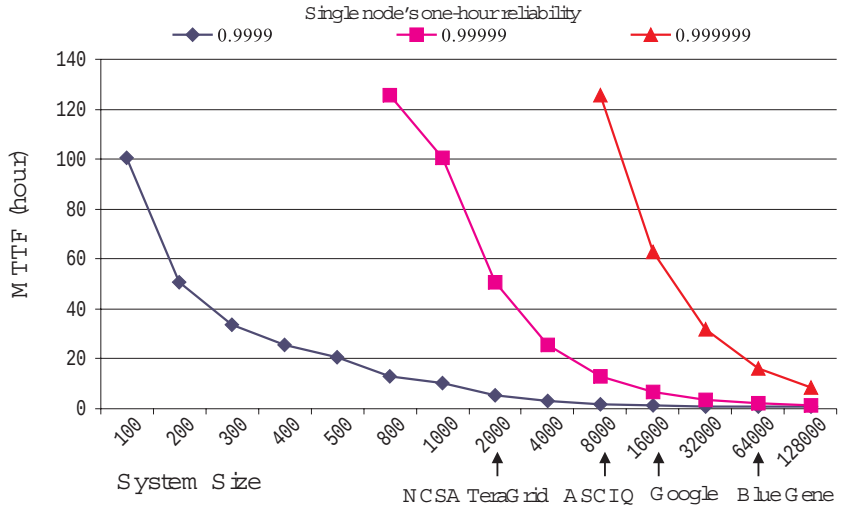


Figure 1.1: Projected system reliability.

The impact of a short MTTF varies for different types of applications. For Internet services such as search engines, Internet portals, and data centers, the failure of a single node degrades performance without bringing down the entire system. Search engines can return fewer query results in the presence of data loss due to failures. Another example is the master-worker computation model, in which a master site distribute work units to client nodes to compute. In this model, client nodes can join and leave freely and the computation can still proceed.

However, most parallel scientific applications are based on the domain decomposition model, in which all processes are equal and each of them holds part of problem domain. In this model, there is a strong data dependence among the participating processes and one process crash is enough to stall the whole computation.

To illustrate this idea, we use the parallel computation of the steady-state heat-flow on a thin slab as an example. The temperature of one point $u(x, y)$ on the slab is described by the Laplace partial-differential equation $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$

To solve this problem by numerical methods, the commonly used approach is finite difference, which discretizes a continuous problem domain into finitely many "cells" and applies some rules

360 teraflops.

to update the cells iteratively. For the above partial-differential equation, the cell update rule is as follows [2]:

$$u(x,y) \leftarrow \frac{1}{4}\left(u(x-1,y) + u(x+1,y) + u(x,y-1) + u(x,y+1)\right)$$

To evaluate the finite difference using parallel programming, we can distribute cells of u evenly among the processes. For simplicity's sake, assume each process has only one cell. At each iteration of evaluation, each process exchange its data with its four neighbors and update the cell it holds, as in Figure 1.2.

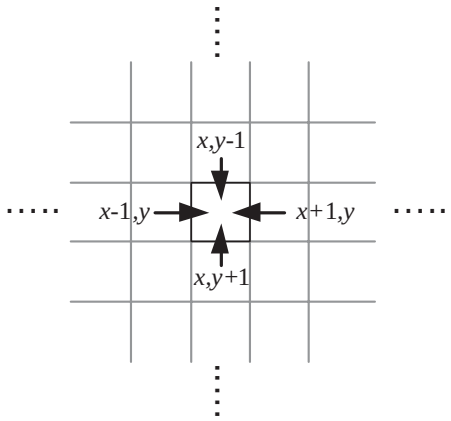


Figure 1.2: Domain decomposition computation.

The dependence of each process on its four neighbors form an interlock, that is, no process can proceed one iteration ahead of or behind other processes. If a process crashes, then effectively this crashed one lags behind, so its neighbors cannot update their cells, and hence the neighbors of these neighbors, and so on. Therefore, one process failure can propagate outward like a ripple and cause the whole computation to halt.

Moreover, the parallel programming toolkit that most scientific applications adopt is the Message Passing Interface (MPI), which offers very little support for fault tolerance and recovery:[2] the default behavior to deal with process failure in most MPI implementations is to abort the program.

[2]We refer to the MPI 1.1 standard [3]. The MPI 2 standard added dynamic process management, allowing processes to be created and removed during run-time, but this standard is not widely adopted.

Once a program is aborted, to resubmit it to the job scheduler could incur long delays in job queues and lead to lost productivity. Therefore, to enable multi-week executions of parallel programs in a large environment composed of commodity hardware, the underlying communication middleware (e.g. the MPI library) must either be able to reconfigure and recover automatically, or provide new function calls that allow programmers to improve fault tolerance.

In addition to MPI itself, another issue in failure recovery is restoring the program to a previous, consistent state. Checkpointing, the action of dumping memory to disk, has been a standard solution used by most scientific applications. But its performance depends on I/O efficiency, which is questionable on large systems, as we explain next.

1.2 Large System I/O

The I/O subsystem in current large systems is insufficient to cope sudden large data flows. Most large systems uses a separate storage farm to serve I/O demands. A small set of nodes in the system act as the conduit to the storage farm. These nodes are usually labeled as I/O nodes and do not participate in any computation. Figure 1.3 illustrates the concept.

Now imagine a massively parallel program which writes tens of gigabytes in one checkpoint

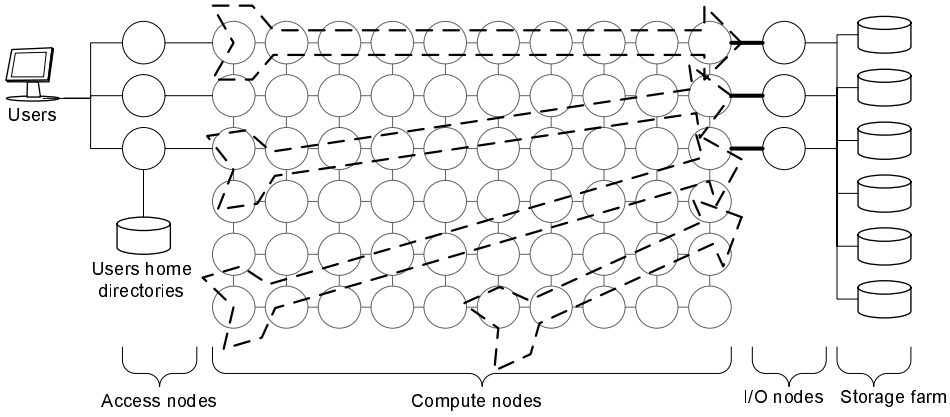


Figure 1.3: Large system I/O. The arrows are I/O writes to the storage farm during a checkpoint.

session.[3] In the current I/O architecture, several places could be overwhelmed by huge inflow data and become a bottleneck: the I/O nodes, their network links to the storage farm and to the rest of nodes, or the storage farm itself. Currently, the state-of-the-art parallel storage solutions can achieve several GB/s of throughput [7]. If there are 10,000 sPPM processes checkpointing and further assume I/O throughput is 5 GB/s, then it requires six to seven minutes to complete a checkpoint session.

1.3 Diskless Checkpointing

Diskless checkpointing is an attractive approach that provides high-performance and reliable storage on large systems. First, a checkpoint is written to memory, which is much faster than disks. To guarantee data integrity, redundancy codes (parity bits or Reed-Solomon codes) are computed and then stored on spares. The spare also assumes the role of compute node if the failed compute node is unable to recover in a short time (e.g. hard failure.) In this sense, diskless checkpointing is akin to software-implemented RAID technology [8]. Finally, I/O is made scalable by partitioning nodes and spares into small groups, and each group takes care of its own redundancy code calculation and node failure and recovery. Figure 1.4 shows the concept of diskless checkpointing on a large system.

We must emphasize that diskless checkpointing is not a replacement of disk-based parallel file systems already running on large systems. Instead, it complements them. The difference is as follows. Diskless checkpointing is meant for *intermediate or temporary data*, especially checkpoint files. It is also specially tailored to MPI programs that perform synchronous checkpointing, i.e. all processes dump to their individual checkpoint files concurrently and there is no sharing among checkpoint files. Disk-based file systems are still essential because users may want to hoard checkpoint files for post-mortem analysis and visualization. Diskless checkpointing just provides a fast storage when checkpoint files are going to be overwritten again and again during the whole

[3] This is not a far-fetched assumption. The sPPM and Sweep3D are two real parallel programs which can dump 200 MB of data *per process* in a checkpoint (see [4, 5, 6] and §6.4).

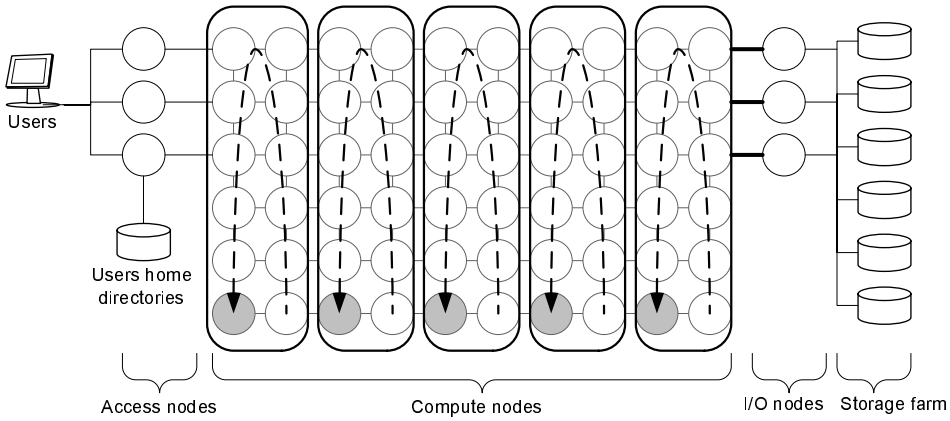


Figure 1.4: Diskless checkpointing on a large system. The compute nodes are partitioned into five groups, each of which has a spare node (gray solid circle). Checkpoint is written to local memory and then the redundancy code over the checkpoint is computed and stored on the spare (hence the direction of the arrow).

execution.

1.4 Thesis Organization

The rest of this thesis is organized as follows. We first examine the failure behavior of high-performance computing systems. We analyzed it from two perspectives: soft (transient) errors (Chapter 2) and hard (permanent) errors (Chapter 3). Results from this study present a realistic view of failure modes and further motivate our goal.

In Chapter 4 we introduce diskless checkpointing and its implementation. We discuss details of data recovery in Chapter 5.

We performed a series of experiments using both benchmarks and two real scientific applications, and the results are presented in Chapter 6. We analyze the results and derive a performance model in Chapter 7.

Chapter 8 and 9 put diskless checkpointing in a broader context. Since diskless checkpointing uses spares, it is possible that spares will be exhausted and the recovery cannot proceed. Therefore in Chapter 8 we develop an analytical model to study both the probability guarantees and time over-

head (combinedly called “performability”) of diskless checkpointing. As a technology projection for large systems, in Chapter (Chapter 9) we present the numerical evaluation of the above model.

In Chapter 10 we survey the related literature. Finally, we discuss directions of future work and conclude this research in Chapter 11.

Chapter 2

Failure Analysis: Soft Errors

Today, clusters built from commodity PCs dominate high-performance computing, with systems containing thousands of processors now being deployed. As node counts for multi-teraflop systems grow to thousands and with proposed petaflop system likely to contain tens of thousands of nodes, the standard assumption that system hardware are fully reliable becomes much less credible.

Hardware failures are usually classified as either hard errors or soft (transient) errors. Soft errors (also known as single-event upsets) include both transient faults in semiconductor devices (e.g., memory or register bit errors) and recoverable errors in other devices (e.g., disk read retries). Conversely, hard errors are permanent physical defects whose repair normally requires component replacement (e.g., a power supply or fan failure).

In this chapter and the next we consider the impact of soft and hard errors and the associated failure behavior. This analysis can provide motivation and insight for designing more reliable systems.

2.1 Introduction

Soft errors are non-repeatable transient faults mostly found in semiconductor devices. In many cases, error detection and recovery mechanisms can mask the occurrence of transient errors. However, on some systems, error detection and correction support may be missing (e.g., due to price-

sensitive marketing of commodity components) or disabled (e.g., for reduced latency on communication channels).

For example, error correcting memory is not used on many consumer PCs, nor are these systems subject to the same level of quality assurance as systems intended for mission critical commercial or scientific domains. Even when the individual systems are well engineered, the multiplicative coupling of large numbers of components can lead to low reliability for the aggregate.

2.1.1 Memory Errors

In an analysis of system logs from workstation clusters, Lin and Siewiorek [9] reported that 90 percent of the crashes were due to soft memory errors. In practice, a single soft memory error rarely causes a system crash, unless it strikes a critical memory region at right time. Hence, the actual frequency of soft errors is higher than that detected – most have no detectable effect.

Improved manufacturing processes and designs and have continued to reduce the hard error rate (HER) for memory modules. Recent estimates range from a mean time before failure (MTBF) of 1,100 years for a 32 Mb DRAM [10] to between 159-713 years for 16 and 64 Mb DRAMs [11, 12]. Overall, the HER has remained roughly constant as memory densities have increased [10].

On the other hand, shrinking geometries, lower voltages, and higher clock frequencies contribute to the growing occurrence of soft errors – the associated decrease in noise margins increases signal sensitivity to transients. Intel reported that the soft error rate for SRAMs increased thirty fold when the process technology shifted from 0.25 to 0.18 micron features and the supply voltage dropped from 2 V to 1.6 V [10].

Soft errors can also arise due to environmental conditions. Poor power regulation and brownouts can induce soft errors because memory cells may not receive enough power to be refreshed. Cosmic rays can also lead to single bit upsets, particularly for systems located at high altitudes. IBM showed that the soft error rate in Denver was ten times higher than that at sea level [13]

Given these diverse conditions, the observed soft error rate (SER) can differ by as much as two orders of magnitude, based on manufacturing process and environmental conditions. Actel

[14] reported that the SER for every Mb of memory manufactured using a 0.13 micron process technology was roughly MTBF of 1-10 years. Tezzaron Semiconductor [15] surveyed recently published data on SER values and concluded that 1000 to 5000 FIT (Failure-In-Time; the number of failures in a billion hours) per Mb was typical for modern memory devices. However, even using a conservative soft error rate (500 FIT/Mb), a system with 1 GB of RAM can expect a soft error every 10 days.

Historically, parity and error correction codes (ECC) have been the primary protection against memory soft errors. SECDEC (Single-Error-Correction, Double-Errors-Detection) is the standard approach, with every 64 data bits protected by a set of 8 check bits. However, ECC does not eliminate all soft errors. Compaq reported that roughly 10 percent of errors are not caught by the on-chip ECC [16].

Constantinescu [17] performed physical fault injection (at IC pin level) experiments to validate the fault/error handling mechanism of the ASCI Red teraflops supercomputer. Stuck-at-0/1 faults were randomly injected at random time instances, to randomly selected signals of data, address, command components of a compute node. The induced errors could simulate many internal faults experienced by the ICs due to environment perturbations. The result showed that 18 percent of errors were uncovered (i.e. uncorrected or undetected) by ECC memory or front side bus.

Moreover, ECC memory solutions generally require 20 percent more die area to fabricate, cost 10-25 percent more, and reduce memory performance by 3-4 percent [15, 18]. In a price sensitive consumer market, these marginal costs are substantial, and many vendors omit these features on consumer-grade products. Therefore, soft memory errors will still be an inevitable reliability problem for future COTS clusters.

2.1.2 Communication Errors

On parallel systems, transient errors can also occur when transmitting messages. Although the MPI 1.1 standard [3] specifies that it is MPI implementor's responsibility to insulate the user from the unreliability of underlying communication fabric, most MPI implementations assume the underlying

communication substrate (e.g., TCP/IP or Myrinet [19]) handles all reliability issues.

However, library or operating system managed end-to-end communication reliability is not without cost – communication latency increases with each software-mediated verification. Indeed, OS-bypass mechanisms, with direct access to network interface cards, were introduced precisely to reduce buffer copying, context switch and interrupt handling overhead [20]. In such situations, message data integrity is dependent on hardware-implemented, link-level checksums.

Stone and Patridge [21] show that link-level checksums are insufficient to detect errors in message. In theory, the chance that link-level checksums do not catch errors should be as small as 1 out of 4 billion packets. After analysis of a trace of 500,000 Ethernet packets that failed TCP's 16-bit checksum, Stone and Patridge found a much higher fraction (1 out of 1,100 to 32,000) should also be caught by link-level checksums but did not.

The source of the errors proved to be host hardware, host software, router memory and links. Indeed, network hardware have been reported to be increasingly susceptible to soft errors [22]. For long-running, communication-intensive codes on large systems, even a small link error rate can have serious implications for application reliability.

2.2 Experimental Methodology

Given the importance of soft errors for both memory and communication systems, we used fault injection techniques to study MPI application responses to transient faults. Fault injection can be either hardware-based or software-based [23]. Each has associated advantages and disadvantages.

Hardware fault injection techniques range from subjecting chips to heavy ion radiation to simulate the effects of alpha particles to inserting a socket between the target chip and the circuit board to simulate stuck-at (e.g., always 0 or 1), open, or more complex logic faults. Although effective, the cost of these techniques is high relative to the components being tested.

In contrast, software-implemented fault injection (SWIFI) does not require expensive equipment and can target specific software components, such as the operating system, software libraries or applications. Therefore, we chose the cost-effective SWIFI to simulate transient errors in memory

and messages during runtime. Below we describe our memory and message fault injection models.

2.2.1 Software Environment

Our experimental target was Intel x86 systems running Linux 2.4, with the MPICH library [24] as the MPI communication toolkit. Software error injection targeted both registers and the application's address space, but not the MPI libraries. The latter was intended to maximize the independence of our results from a specific MPI implementation.

To inject faults, we linked the target applications with a custom fault injection library containing MPI wrapper functions. Each wrapper performs fault injection tasks and then calls the actual MPI function via the MPI profiling interface (PMPI).

```
int MPI_Init( int * argc,char *** argv) {
  <performs some fault injection tasks>

  PMPI_Init(argc, argv);
}
```

Our `MPI_Init()` wrapper, shown above, parses a configuration file and spawns the memory fault injector. The fault injector awakens periodically and invokes the `ptrace()` UNIX system call to halt the target process and overwrite target process memory or register content to simulate the effect of transient errors. The target process is then allowed to resume execution and its reaction to faults is recorded.

2.2.2 Memory Fault Injection

Memory fault injection targeted both registers and applicaton memory regions. All registers (including regular and x87 floating-point ones) were targeted except the following: system control (CR0-CR4), debug and performance monitoring (DR0-DR7 and MSRs) and virtual memory management (GDTR, LDTR, IDTR, and TR). Modifications to these registers can cause system crashes, complicating application experiments. We also omitted the TLB and the L1 and L2 caches. Mod-

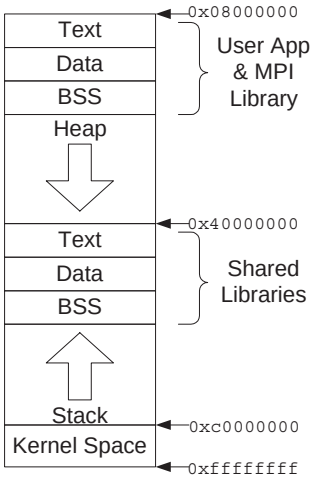


Figure 2.1: Linux process memory model.

ifying the latter would have required a kernel implementation, something we sought to avoid for platform portability.

As we noted above, the memory region where we injected faults was confined to the address space of an MPI process: the text, stack and heap, as shown in Figure 2.1. We excluded other portions of the memory because we wanted our results to be independent of the execution context, and we wanted to maximize the probability of application error effects. Injecting transient faults into unused memory has little effect on applications.

To selectively inject faults into a user application's context and not the MPI library, our fault injector employs different techniques for different regions in the address space.

Text, Data and BSS. The identity and location of text, data and BSS memory objects are determined at compile time and are static. To separate the MPI library's memory objects from the user application's, we processed the library and application binaries to retrieve the respective lists of {symbolic name, address} pairs. We then constructed a fault dictionary containing several thousand addresses randomly selected from this list. Any address whose associated symbolic name also appears in the MPI library's list was removed as a possible injection point.

Heap. The heap stores data structures whose memory is dynamically allocated at runtime

(i.e., by `malloc`, `realloc` and `free` in C and similar calls in C++). To identify the heap area allocated with the MPI library, we implemented a customized memory allocator that wraps the standard `malloc` using the GNU C library's "memory allocation hooks." Then we were able to keep track of memory objects allocated by the application and modify its content during run-time.

Stack. Like the heap, the stack also resizes dynamically. In the Intel x86 architecture, the stack is composed of stack frames. Each function call pushes a frame onto stack, and each return pops a frame. Each frame contains saved registers, arguments, local variables, return address, and a pointer to the next frame.

The stack frames in use by an application can be identified by a walk-through from the top to bottom frames (using the EBP and ESP registers) and by examination of the "return address" field in each frame. If the return address falls within user application's text region, then the frame immediately below is in user application's context and is subject to our fault injection.

2.2.3 Message Fault Injection

For MPI message injections, we modified the payload received immediately from the underlying communication software, as shown in Figure 2.2. MPICH is implemented in three layers: (a) API, which connects the MPICH library to the user application, (b) ADI (Abstract Device Interface), which implements MPI functionality at a network-independent level and (c) Channel, which is the interface between MPICH and the underlying network-specific communication software.

We configured MPICH to use the `ch_p4` channel and injected faults at the Channel level. We chose to inject the faults into incoming traffic immediately after MPICH invokes the `recv()` UNIX socket routine. Although TCP/IP checksums, coupled with link-level CRCs, are very effective in preventing data corruption, our purpose was to simulate the effects of soft errors that are undetected in the transmission path when only a link-level CRC is present.

In reality, message errors can also originate from network hardware or operating systems. However, injecting faults there either requires special equipment or can cause instability. The fault injection process will also be more time consuming; after each injection, the system must be re-

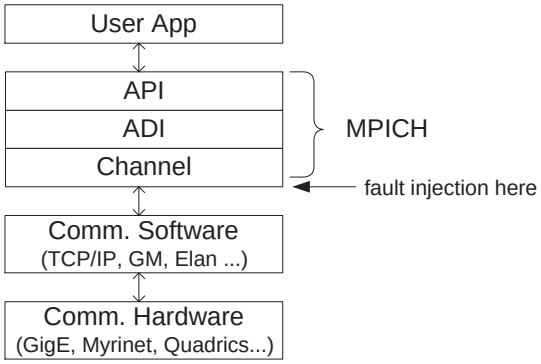


Figure 2.2: Fault injection for MPI messages

booted to restore to a clean state. Because the systems on which we conducted tests are also used by others, operating system fault injection was eliminated.

Before performing message injections, we profiled the application to estimate the total message volume received by each MPI process during the execution. During each injection experiment, we generated a uniform random number in this range. The modified MPICH library maintains a counter on received message volume and overwrites the payload when the counter value coincides with the random number.

2.3 Experimental Environment

The hardware experimental environment is a meta-cluster formed from two Linux PC clusters. The first cluster (Rhapsody) has 32 nodes connected by both 10/100 and Gigabit Ethernet. Each node has dual 930 MHz Pentium III processors and 1 GB of DRAM. The second, older cluster (Symphony) has 16 nodes connected by Ethernet and Myrinet; each node has dual 500 MHz Pentium II processors and 512 MB of RAM.

2.3.1 Test Applications

We used three scientific codes as test applications: Cactus Wavetoy [25], NAMD [26] and CAM [27]. To reduce the time needed to conduct experiments to tractable levels, we modified each appli-

	Cactus Wavetoy		NAMD		CAM	
Memory (MB)	1.1		25-30		80	
Text Size	0.3		2		2	
Data Size	0.13		0.11		32	
BSS Size	$\ll 0.1$		0.6		38	
Heap Size	0.45-0.5		22-27		8	
Message (MB)	2.4-4.8		13-33		125-150	
Distribution	Header	User	Header	User	Header	User
Percentage	6	94	8	92	63	37

Table 2.1: Per-process profiles of test applications

cation's input parameters such that each executed for only for 2-5 minutes.

However, we ensured that each application executive several phases (i.e. loop iterations or time steps), as would be typical of normal execution. Despite these parameter modifications, the injection experiment consumed two months of time on the two target clusters.

We profiled three test applications to quantify their memory use and communication frequency and volume. The purpose of profiling was to provide a baseline for interpreting the experimental results and to explain the error behavior. Table 2.1 shows the per-process application profiles.

Cactus Wavetoy

Cactus [25] is a modular toolkit for developing scientific codes. Wavetoy is a test program from the Cactus package that solves hyperbolic PDEs. For our fault injection experiments, we used a problem size of 150x150x150 and 100 steps. At the end of an execution, the process of rank 0 writes the application results to output files in plain text format. For each execution, Wavetoy spawns 196 MPI processes, each processor serves two MPI processes, and the application executes for just under one minute.

NAMD

NAMD [26] is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. We used a 92,000 atom "apoa1" input problem, whose data size was

Region	Executions	Errors (%)	Error Manifestations (%)		
			Crash	Hang	Incorrect
Regular Reg.	508	62.8	44	56	
FP Reg.	500	4.0		50	50
BSS	502	6.2	19	81	
Data	500	2.4	50	50	
Stack	980	12.7	65	35	
Text	1000	6.7	73	18	9
Heap	933	5.0	8	72	20
Message	2000	3.1	26	42	32

Table 2.2: Fault injection results of Cactus Wavetoy

Region	Executions	Errors (%)	Error Manifestations (%)				
			Crash	Hang	Incorrect	App. Det.	MPI Det.
Regular Reg.	498	38.5	86	10	4		
FP Reg.	500	7.6	39	11	3	47	
BSS	497	1.8	78	22			
Data	502	4.2	95			5	
Stack	493	9.3	74	13	6		6
Text	498	8.4	79	7	7	7	
Heap	500	5.2	81	8	8	3	
Message	500	38.0	26		28	46	

Table 2.3: Fault injection results of NAMD

20 MB. Each NAMD execution spawned 96 MPI processes and executed for 2.8 minutes.

As a baseline for output comparison, we used the NAMD console output, which shows total energies, temperature and pressures at each time step. In NAMD, each MPI process holds a portion of the input set of atoms. Each time step updates the atoms' positions and velocities. However, the order that these updates occur depends on the MPI message arrival order.

As such, NAMD executions are nondeterministic, and the output files can differ across executions. The only reproducible output is the console output, which has no noticeable deviation if the number of steps is less than 20, which we used in our experiments.

Region	Executions	Errors (%)	Error Manifestations (%)				
			Crash	Hang	Incorrect	App. Det.	MPI Det.
Regular Reg.	500	41.8	68	26	5	1	
FP Reg.	422	8.0	33	15	26	26	
BSS	500	3.2	62	25	13		
Data	500	2.8	50	50			
Stack	500	6.2	71	10		13	6
Text	500	14.8	78	11	7	4	
Heap	500	2.6	31	69			
Message	500	24.2	21	4	71	3	

Table 2.4: Fault injection results of CAM

CAM

The Community Atmosphere Model (CAM) [27] is the atmospheric component of a larger, global climate simulation package called CCSM, the Community Climate System Model. In our experiments, we used CAM version 2.0.2 with the default test data sets and initial condition files as input, totalling 96 MB.

Each CAM execution used 64 MPI processes. The input data specified 24 hours of simulated time and took 4 minutes of execution to complete. The 76 MB of output is written to disk by the process of rank 0 at the end.

2.4 Experimental Results

We executed each of the three applications (Cactus, NAMD and CAM) on our test clusters with the fault injection methodology described in §2.2. From these experiments, we calculated the error rate, which is the ratio of manifestations to injected faults. For all manifested faults, we also observed the error manifestations and calculated the ratios of different manifestations. Before analyzing the results, we summarize the range of error manifestations.

2.4.1 Error Manifestations

If the injected fault does not manifest, we labeled the outcome as correct. Otherwise, the induced errors are categorized into several disjoint classes.

Crash. Application crashes were detected by identifying MPICH error messages in the STDERR output. MPICH handles all critical signals (e.g., SIGSEGV and SIGBUS) due to abnormal termination of both the user application and itself.

Hang. Because our experimental environment was under our exclusive control, there was little variability in execution times. Hence, for each application execution, we waited for one minute beyond the expected execution completion time. If the application did not complete during this time, we terminated the application and labeled the outcome as an application hang.

Application Detected. Some of the applications in our suite implement internal consistency checks. After a consistency failure, these applications print error messages to console and abort. Therefore, by examining the console output, we identified such errors.

MPI Detected. The MPI 1.1 standard specifies that by default, an error during the execution of an MPI call causes the application to abort. However, MPI provides mechanisms for users to handle recoverable errors by registering customized error handlers via the `MPI_Errhandler_set` call. Therefore, we registered such a handler, and whenever the handler was invoked, the handler labeled the outcome as "MPI detected."

Incorrect Output. After each execution, we compared the application output against the correct one to test for silent data corruption. We labeled the outcome of an injection as incorrect if the user application finishes execution without reporting an error, but the output was incorrect. This is most dangerous of all possible errors because there is little sign during the execution that can alert the user.

2.4.2 Results

Table 2.2 summarizes the results for Cactus Wavetoy. During our tests, no Application Detected or MPI Detected errors were encountered. Table 2.3 and 2.4 show the results for NAMD and CAM,

respectively.

2.5 Analysis

2.5.1 Register Injections

Even a cursory examination of Tables 2.2-2.4 shows that the regular (integer) registers are the most vulnerable to transient errors, with an error rate ranging from 38.5 to 62.8 percent. Because the Intel x86 architecture has less than a dozen general-purpose registers, most contain live data at any given time. Single bit upsets in these registers are very likely to affect application behavior.

These effects are strongly dependent, however, on the quality of live register allocation and management (a function of the compiler) and the size of the register file. One would expect different sensitivity on systems with a larger register file. For example, Springer [28] investigated the register usage of an image processing kernel on a PowerPC 750 system and found that only 4-5 of 64 available registers were used during execution. If the code were compiled with the optimization switch `-O`, then the number of live registers jumped to 14-15. The suggests that a program could be made more robust if it is compiled without register optimizations, albeit with possible performance loss.

The error rate for floating-point register fault injection is much lower than that for integer registers, with only a 4-8 percent error rate. There are several possible reasons for this low error rate.

The Intel x87 FPU has seven special-purpose registers (CWD, SWD, TWD, FIP, FCS, FOO, and FOS) and eight FPU data registers, which are placeholders for floating-point numbers [29].

First, we found that most special-purpose register injections did not induce errors, except for the TWD register, which will possibly cause NaN (Not a Number) errors. The TWD (tag word) register indicates the content of each of the eight FPU data registers. The content can be a valid number, zero, special (NaN, infinity, or denormal,) or empty. Changing one bit can turn a valid number into NaN or zero.

Second, the x87 FPU instructions treat the FPU data registers as a register stack, which is

addressed relative to the register atop the stack. For the three applications we tested, we examined their assembly codes generated by the compiler and found that no more than four, and usually only two, FPU data registers are used.

Finally, because the FPU data registers are 80 bits long, based on the IEEE floating point standard, some bits are rounded when the value in an FPU data register is written to memory. When combined, these three factors can cause the low observed error rates.

2.5.2 Memory Injections

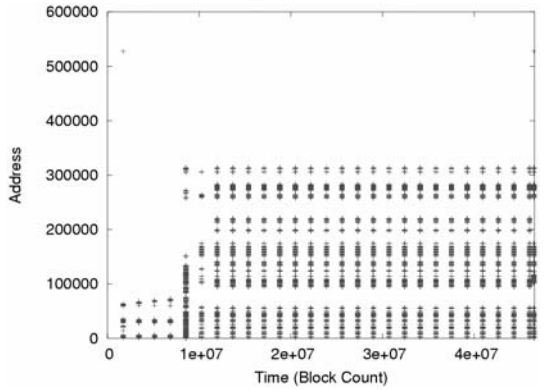
The error rates for memory injections were consistently low, generally less than 10 percent, across the three applications. Table 2.1 also suggests that the error rate is largely independent of memory region size. For example, the data section sizes range widely from 130 KB to 38 MB, yet the error rates vary only between 2.4 and 4.2 percent.

Given temporal and spatial locality, we conjectured that either most of the memory is never accessed (i.e. faults are not within the spatial locality), or the faults are injected into memory locations that will not be accessed again or will be overwritten before accessed (i.e. faults are not within the temporal locality.)

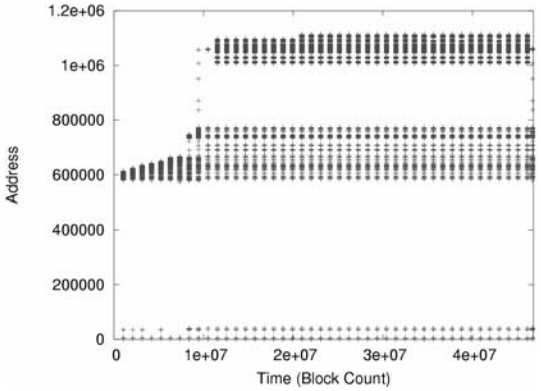
To verify this conjecture, we used the open-source memory debugging tool Valgrind [30] to trace the memory accesses of the three applications. Valgrind works directly on executable binaries and can instrument each x86 instruction. We used Valgrind to collect the following run-time memory access data: text accesses, which are executed instructions, and data accesses, which are memory loads in Data, BSS, and Heap sections.[1] We recorded snapshots of text and data accesses periodically to understand temporal and spatial locality and their relation to error rates.

Tables 2.5–2.7 show the results of these measurements. The memory address shown is the address relative to the beginning of the respective sections. Due to instrumentation overhead, the

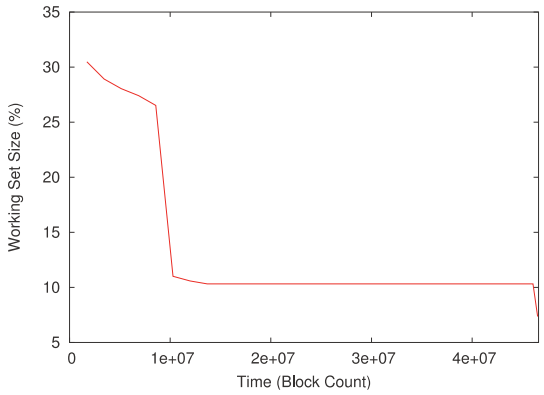
[1]For measurement simplicity, this data is drawn from instrumentation of a randomly selected MPI process, with the application executed on a smaller number of processors. Given the characteristics of our application suite, we believe this data is representative.



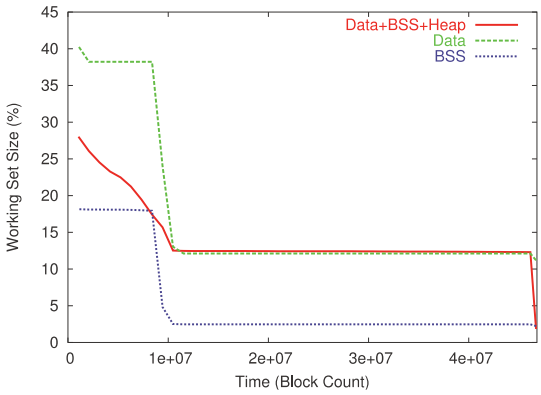
Text accesses



Data+BSS+Heap loads



Text working set



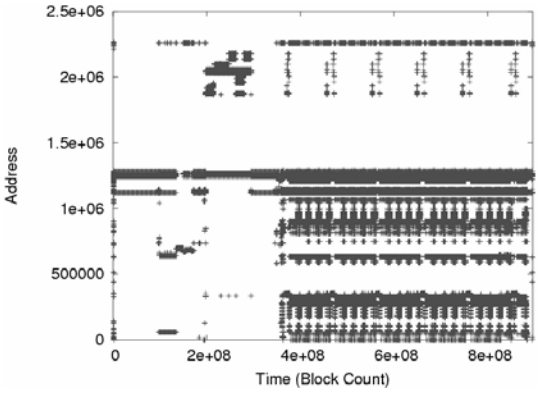
Data+BSS+Heap working set

Table 2.5: Memory Trace of Cactus Wavetoy

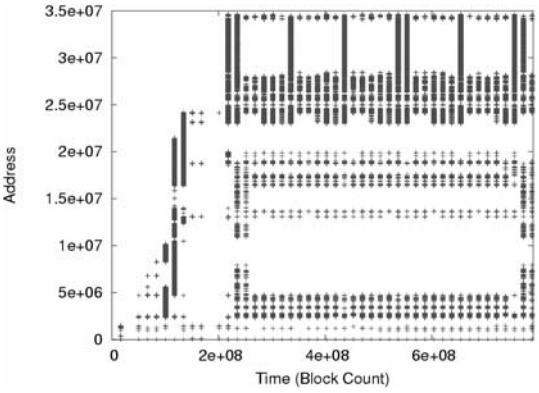
applications run 2 to 5 times slower than normal. To establish a consistent time frame across executions, we used the basic block count to measure the elapsed time.

Because injecting faults into unused memory has no effect, it is crucial to identify how much memory is actually accessed. To estimate this, we calculated the working set size, where the "working set size at time t" is the size of accessed memory since t. The working set size, therefore, is a non-increasing function of t. To relate the error rate with the working set size, we plotted the percentage of working set size relative to the respective section sizes in Tables 2.5–2.7.

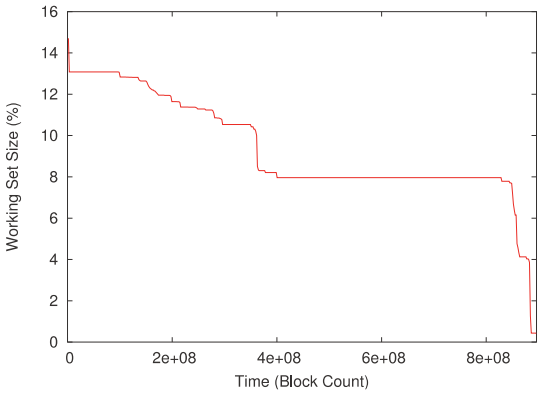
We must emphasize that there are also scientific applications with large memory footprint and large working set size. For these kind of applications, the error rates of memory injection could be much higher.



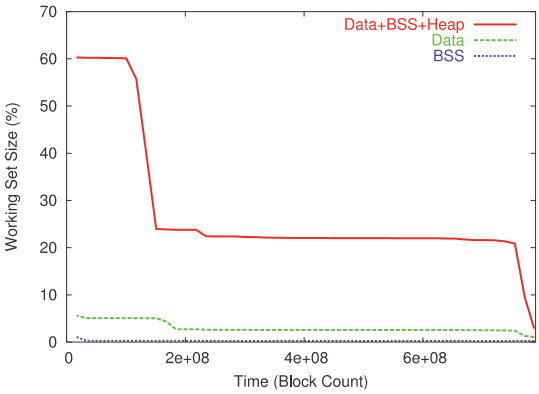
Text accesses



Data+BSS+Heap loads

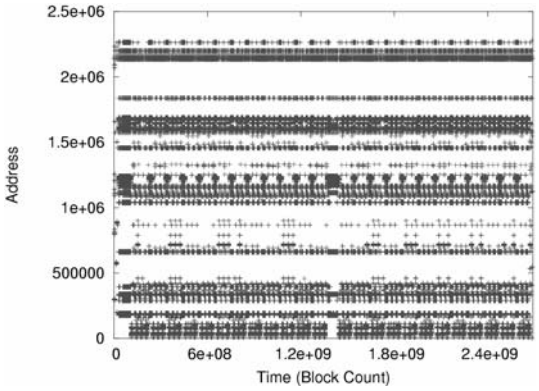


Text working set

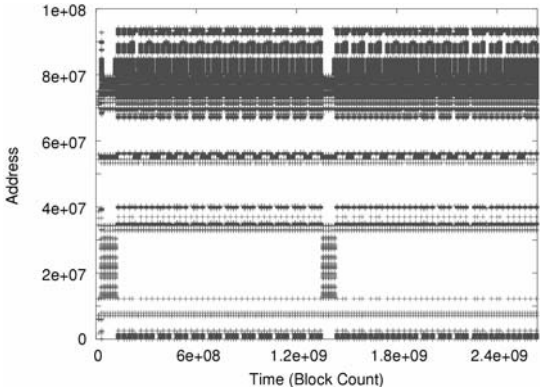


Data+BSS+Heap working set

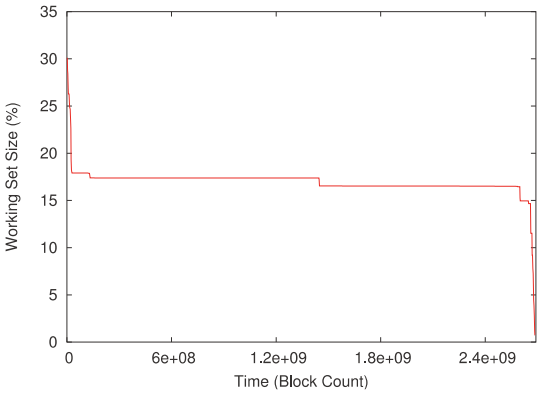
Table 2.6: Memory Trace of NAMD



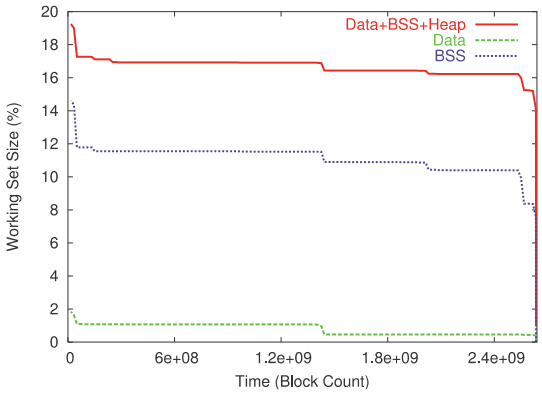
Text accesses



Data+BSS+Heap loads



Text working set



Data+BSS+Heap working set

Table 2.7: Memory Trace of CAM

Based on this data, the following observations are apt. First, all three applications exhibit phase behavior in their memory accesses: the initialization phase and the computation phase. The phase shift occurs when there is a large drop in working set size because the working set has moved from startup code to the computation kernel (spatial locality). During the computation phase, memory accesses are very periodic and regular (temporal locality), and the working set remains unchanged.

Second, the working set size plots suggest the cause of the low error rate from fault injections. For the text section, the working set size at time 0 is 30 percent for Cactus and CAM and 15 percent for NAMD. Entering the computation phase, the working set size declines to 10, 8 and 13 percent for Cactus, NAMD, and CAM, respectively. Compared to the text injection error rates, which are 6.7, 8.4, and 14.8 percent, the small working set size is the cause of the low error rates. Our results are consistent with [28], where only a fraction of the heap was found to be used.

The working set analysis also shows that most memory in the Data, BSS, and Heap area is either not accessed at all or is not accessed after the initialization phase. At time 0, the Data+BSS+Heap working set size is 28, 60, and 19 percent for Cactus, NAMD, and CAM, respectively. During the computation phase, this size drops to only 12, 22, and 16 percent. A close look at the Data and BSS sections shows that their working sets are usually even smaller, mostly less than 10 percent. These results strongly correlate with the low error rates in Data+BSS+Heap injections.

Unlike the text section, the working set alone cannot completely explain the error rates for Data+BSS+Heap injections; the text is read-only, whereas Data+BSS+Heap can have many interleaving writes and reads. Although we did not record the most recent write to each memory location before read, we conjecture that a corrupted memory cell is overwritten by the application before it is loaded and used again. In addition, a bit error in the instruction opcode can alter the instruction and halt the execution, whereas a bit error in the data could be more innocuous. We believe this can also lead to low error rates in Data, BSS, and Heap areas.

2.5.3 Message Injections

Since messages will certainly be read by the receiver, we expect the message injections to have a higher error rate. Indeed, we observed that both NAMD and CAM are quite sensitive to message injections, with 38 and 24 percent error rates, respectively.

The difference between NAMD and CAM is that NAMD can detect 46 percent of these errors, while CAM only detect three percent of them. As with floating point errors, we attribute NAMD's high detection rate to its built-in message consistency checks, which CAM lacks. An instrumentation of NAMD code shows that these internal checks increases the execution time by three percent, but can detect many errors.

We also observed a few "MPI Detected" error manifestations for NAMD and CAM. All were associated with memory errors in the stack contents. Recall that MPI allows the user application to register error handler callback functions. However, in MPICH, the callback is triggered only when incorrect arguments are passed to MPI routines (e.g., a non-existent destination specified for a send operation). Stack error injections trigger such errors because the stack holds the arguments to function calls. Other errors, such as abnormal termination of an MPI process due to fault injection, do not trigger the error handler. Instead, MPICH itself will abort the user application, which we labeled as an application Crash.

The MPI 1.1 standard gives implementors considerable liberty concerning those errors that can raise the error handler. To assess alternatives, we examined the source code for two other popular MPI 1.1 implementations, LAM/MPI [31] and LA-MPI [32]. We found that they also only raise the user-registered error handler when argument checks fail. Abnormal termination of peer MPI processes will abort the application without invoking the error handler, just as MPICH does.

The error rate for Cactus is only 3.1 percent, much lower than NAMD and CAM. We found this seemingly counterintuitive phenomenon is due to several factors: the MPICH traffic structure, Cactus's message passing behavior, and Cactus's output format.

Recall from §2.3.1 and Table 2.1 that the MPICH traffic can be roughly classified as header and user data. For Cactus, 94 percent of its incoming MPI traffic is user data and 6 percent consisted

of headers only. A substantial fraction of Cactus data transfers are large arrays of floating-point numbers, whose perturbation does not crash or hang the application; rather, these errors are manifest in other ways. In contrast, perturbing the headers has about a 40 percent probability of corrupting the Cactus execution. Therefore, the combined Crash and Hang rate is 6*0.4 or roughly 2.4 percent.

Because user data is the majority of Cactus message traffic, message fault injection should induce many cases of incorrect output. However, we found experimentally that this was not true. The reason is the output data representation. As mentioned in §2.3.1, we configured Cactus Wavetoy to write its output textually, which has the advantage of portability. Platform differences such as byte order are avoided. However, for Cactus Wavetoy, it hides small changes in low order decimal digits.

A detailed examination of Cactus message data showed that most transferred data are very close to zero. Only when faults occur in the significant bits of the exponent or mantissa will the output be incorrect. We also noted that executing more Cactus Wavetoy iterations will almost always yield incorrect outputs (i.e. the error amplifies as the computation continues). A binary output format would detect more cases of incorrect output.

2.6 Summary

From our experiments, one can draw several conclusions. First and most importantly, soft bit errors can adversely affect application reliability on commodity parallel systems. Their impact is proportional to the size of memory accessed by the application. Without hardware checksums, ECC memory and application-specific error checking, soft errors, particularly on large systems, will trigger application crashes, hangs or incorrect results.

The definition of correctness is also often application specific, and different definitions could lead to different error manifestation results. For Cactus Wavetoy, results presented in plain-text format have lower precision and can mask some injected faults. In turn, NAMD execution is non-deterministic, making error identification difficult. The distinction between memory fault induced errors and small variations due to numerical roundoff are subtle and difficult to detect.

In detecting communication errors, NAMD's message checksum is effective at low cost – only

three percent overhead. However, NAMD's checksum only tests user data, not headers, which can only be observed inside the MPI library. If an application transfers a larger volume of user data per unit time, the overhead for application-level message checksums can rise substantially.

Program assertions and sanity/consistency checks are usually used for debugging and are removed in production code. In our experiments, they can also capture some of injected faults. Use of internal checks is an important aspect of robust application implementation, but must be used wisely because excessive checks can still harm performance.

Chapter 3

Failure Analysis: Hard Errors

In the preceding chapter we analyzed the soft errors in memory and communication and their impact to user applications. In this chapter we continue to explore the case of hard errors. Hard errors refer to the permanent physical defects whose repair normally requires component replacement, such as a power supply or fan failure.

Compared to soft errors, hard errors are much less likely to occur, but their impact is immediate and more influential. This is because soft errors sometimes can be masked by error detection and correction mechanisms, or they do not cause any harm at all. On the other hand, applications are bound to crash if there is any hard error of any kind. Thus, the analysis of hard errors is as crucial as that of soft errors.

Since the application failure behavior due to hard errors is simple and well-understood, our study of hard errors is to understand their pattern, such as frequency and correlation. To this goal, we obtained and analyzed the failure data of three high-performance computing systems.

3.1 Terminology

Certain terms in reliability theory are often used in confusion in the literature, and a clarification is needed here. The Time to Failure (TTF) is the interval between the end of the last failure and the beginning of the next failure. The Time between Failure (TBF) or the Time between Interruption

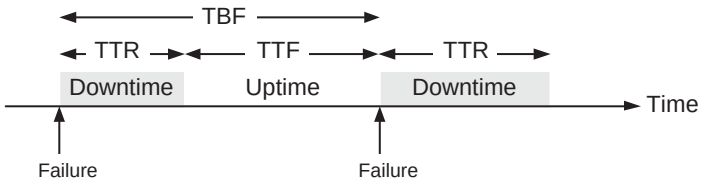


Figure 3.1: TTF, TBF, and TTR.

(TBI) are the interval between the beginnings of two consecutive failures. The Time to Repair (TTR) is the duration the repair takes place.

Figure 3.1 illustrates these ideas. In general, TTF is used for a system that is either in working or in failure mode, e.g. a single node. TBF is used for a system consisting of multiple components, each of which is either in working or in failure mode, and the system is consider operational so long as at least one components is still working, e.g. a PC cluster. In Figure 3.1, for example, the two failures could be failures of two different nodes in a cluster.

Reliability refers to how long a component works before it fails. It is usually described in terms of Mean Time to Failure (MTTF). Availability is the ratio of time the component is working, i.e. the ratio of uptime to total time (or MTTF / (MTTF+MTTR)). For multi-node systems, the availability of the entire system is the average of node availability.

3.2 Systems and Measurements

We obtained the failure log of three systems at the National Center for Supercomputing Applications (NCSA). These three systems are quite different architecturally. The first is an array of SGI Origin 2000 (O2K) machines. SGI Origin 2000 is a cc-NUMA distributed shared memory supercomputer. An O2K can have up to 512 (and usually 128-256) CPUs and 1 TB of memory, all under control of a single-system-image IRIX operating system. The SGI Origin 2000 system installed at NCSA is an array of twelve O2K's (total 1,520 CPUs) connected by proprietary, high-speed HIPPI switches. Table 3.1 lists the CPU count and memory size of individual machines. The machines A, B, E, F, and N are equipped with 250 MHz MIPS R10000 processors, and the rest with 195 MHz MIPS

R10000 processors. M4 accepts interactive access, while the others machines only service batch jobs. Peak performance of NCSA O2K is 328 gigaflops.

The second and the third systems are Beowulf-style PC clusters. The "Platinum" cluster has 520 two-way SMP 1 GHz Pentium-III nodes (1040 CPUs), 512 of which are compute nodes (2 GB memory), and the rest are storage nodes and interactive access nodes (1.5 GB memory). The "Titan" cluster consists of 162 two-way SMP 800 MHz Itanium-1 nodes (324 CPUs), 160 of which are compute nodes (1.5 GB memory) and 2 are for interactive access. Both clusters use Myrinet 2000 and Gigabit Ethernet as system interconnect. Myrinet is faster and for node communications, whereas the Gigabit Ethernet is slower and serves I/O traffic. Both clusters have one teraflop of peak performance.

All three systems use batch job control software to manage workload. The O2K uses the Load Sharing Facility queueing system. Each job on the O2K have resource limits of 50 hours of run-time and 256 CPUs. The Platinum and the Titan employ the Portable Batch System with the Maui Scheduler, and the job limits are 352 and 128 nodes for 24 hours, respectively.

The failure log was collected in the form of monthly or quarterly reliability reports, as shown in Figure 3.3. At the end of a month or a quarter, a report for each node/machine is created. A report records outage date (but no outage time), type, and duration. There are five outage types defined by NCSA system administrators: Software Halt (SW), Hardware Halt (HW), Scheduled Maintenance (M), Network Outages, and Air Conditioning or Power Halts (PWR). The cause of an outage is determined as follows: a program runs at machine boot time prompts the administrator to enter the reason for the outage. If nothing is entered after two minutes, the program defaults to recording a Software Halt. It is possible that some of the recorded Software Halts may be actually transient hardware faults, but the absence of a more detailed log prevented us from further investigation.

The data collection period is two years (April 2000 to March 2002) for the O2K and eight months (January 2003 to August 2003) for the Platinum and the Titan. In this set of failure log, there is no occurrence of Network Outage, so we exclude it from the rest of analysis.

3.3 Results

We summarize the failure data in Table 3.1 and 3.2, and Figure 3.2. There are two kind of availability measures. The overall availability is computed as

$$1 - \frac{\sum(\text{\# Down CPUs} \times \text{Downtime})}{\text{\# Total CPU} \times \text{Total time}}$$

The scheduled availability removes the Scheduled Maintenance (M) downtime from consideration and only counts scheduled uptime as total time, so it is computed as

$$1 - \frac{\sum(\text{\# Down CPUs} \times \text{Unsched. Downtime})}{\text{\# Total CPUs} \times \text{Sched. time}}$$

Note that in the O2K's case, the twelve machines have different number of CPUs, so "# Down CPU" is the number of CPUs on the failed machines.

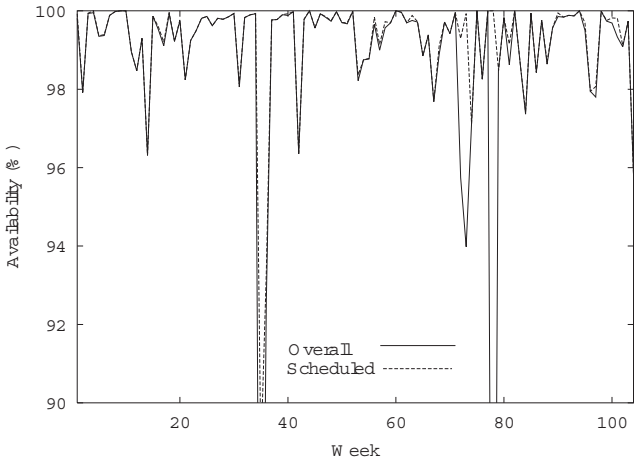
Because the failure log does not include the start and end times of outages, we can only calculate TBFs in terms of days. For the whole system of the O2K, the TTF reported in Table 3.1 is TBF, and the downtime is the weighted average of individual machine downtimes:

$$\frac{\sum(\text{\# Down CPUs} \times \text{Downtime})}{\text{\# Total CPUs}}$$

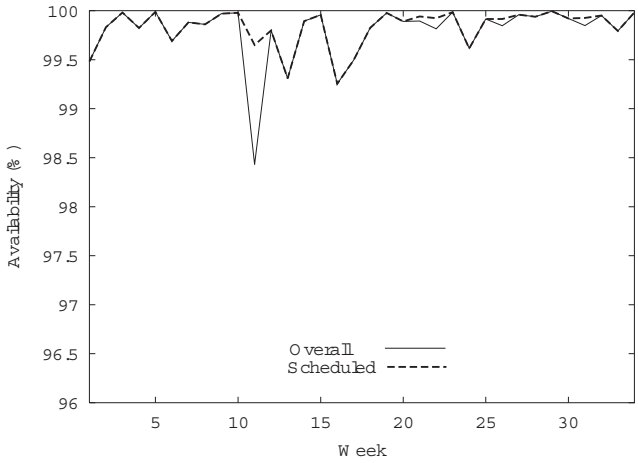
From the data, it is apparent that software halts account for most outages (59-83%), but the average downtime (i.e. MTTR) is only 0.6-1.5 hours. On the other hand, although the fraction of hardware outages is meager (1-13%), the average hardware downtime is the greatest among all unscheduled outage types (6.3-100.7 hours). This is reasonable because hardware problems usually require replacing parts and performing tests, whereas many software problems can be fixed by reboot.

We contacted the NCSA staff about the exact causes of software and hardware halts. We were told that for the Platinum and the Titan, there were two or three cases where power supplies needed to be replaced; otherwise, the main cause of hardware outages is the Myrinet, including network cards, cables, and switch cards.

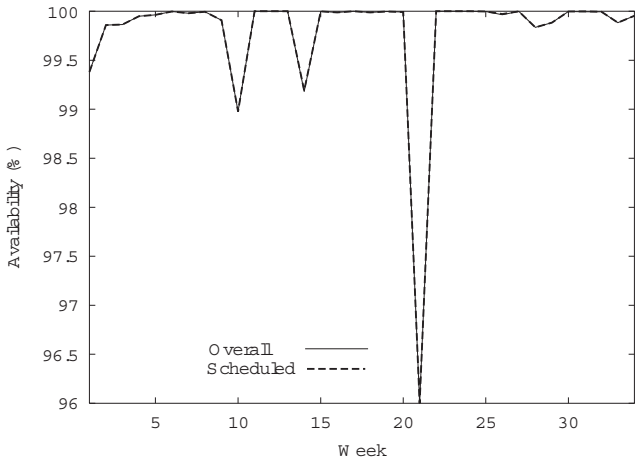
The physical Myrinet network structure is as follows. A network card resides at a host PC and is connected by cables to the Myrinet switch enclosure. A Myrinet switch enclosure stacks



O2K



Platinum



Titan

Figure 3.2: NCSA systems availability.

	All	A	B	E	F	H1	H	J	M	M2	M4	N	S
CPUs	1520	128	256	128	128	128	128	128	128	64	48	128	128
Mem (GB)	618	64	128	64	76	64	32	32	32	16	14	64	32
Outages	687	87	182	40	81	42	25	32	24	41	59	37	37
SW (%)	59	74	68	53	63	57	60	59	42	44	39	49	46
HW(%)	13	8	19	13	9	21	8	3	17	10	5	5	32
M(%)	21	11	12	28	20	12	24	19	29	32	49	38	14
PWR(%)	7	7	2	8	9	10	8	19	13	15	7	8	8
Downtime (day)	9.5	8.7	19.2	15.3	5.9	13.8	6.2	3.6	4.8	7.5	4.5	5.5	5.0
SW (%)	27	32	49	11	36	9	5	25	15	27	12	6	16
HW(%)	28	35	29	1	10	67	49	< 1	22	12	4	28	22
M(%)	41	28	20	85	44	22	45	66	58	52	75	62	58
PWR(%)	4	4	2	3	9	2	2	9	4	9	10	4	4
Avail(%)	98.7	98.8	97.4	97.9	99.2	98.1	99.2	99.5	99.3	99.0	99.4	99.2	99.3
Avail2(%)	99.2	99.1	97.9	99.7	99.6	98.5	99.5	99.8	99.7	99.5	99.8	99.7	99.7
MTBF (day)	1.0	8.1	4.0	15.9	8.6	14.7	29.5	22.5	30.3	17.4	12.3	18.6	18.5
StdDev	2.1	14.5	5.8	28.5	14.7	20.9	36.9	33.4	48.3	31.7	18.7	34.3	22.9
50 %	0.9	1.7	2.1	1.7	2.5	3.5	25.0	1.0	4.0	1.6	5.7	1.0	11.0
75 %	1.9	8.8	5.5	20.3	10.0	18.8	46.0	44.3	34.5	28.0	15.8	29.0	28.5
90 %	3.8	27.4	11.7	50.1	30.0	49.9	69.6	70.9	88.8	77.0	33.2	64.4	56.4
MTTR (hr)	3.5	2.4	2.5	9.2	1.7	7.9	6.0	2.7	4.8	4.4	1.9	3.6	3.2
StdDev	13.1	7.8	5.2	29.9	5.1	33.2	15.6	7.5	9.3	7.7	8.1	8.8	7.2
50 %	0.5	0.4	0.9	0.43	0.4	0.5	0.5	0.3	0.5	0.7	0.4	0.4	0.5
75 %	1.5	1.0	2.0	2.5	0.7	1.6	1.4	0.5	1.9	4.9	0.8	1.2	1.3
90 %	8.4	3.3	6.4	16.0	3.5	14.9	14.8	3.9	18.8	13.5	1.8	13.7	10.7
MTTR SW	1.5	1.1	1.8	1.9	1.0	1.3	0.5	1.1	1.7	2.7	0.6	0.4	1.1
MTTR HW	6.3	10.5	3.9	0.6	2.0	24.7	36.3	0.4	6.4	5.4	1.3	18.6	2.2
MTTR M	8.0	5.8	4.3	28.6	3.9	14.6	11.2	9.6	9.6	7.1	2.8	5.9	13.8
MTTR PWR	2.1	1.5	3.4	3.8	1.0	1.5	1.3	1.2	1.7	2.8	2.6	1.7	1.7

Table 3.1: NCSA Origin 2000 failure summary. SW=Software Halts. HW=Hardware Halts. M=Scheduled Maintenance. PWR=Air Conditioning or Power Halts. Avail is the overall availability, and Avail2 is scheduled availability. 50% means 50 percentile.

	Platinum	Titan
Outages	7279	947
Outage/Node	14.00	5.85
SW (%)	84	60
HW (%)	< 0.1	5
M (%)	16	1
PWR (%)	0	34
Downtime/Node (hr)	12.16	12.55
SW (%)	69	18
HW (%)	10	18
M (%)	21	< 1
PWR (%)	0	64
Avail (%)	99.79	99.78
S Avail (%)	99.83	99.79

	Platinum	Titan
System MTBF (hr)	0.79	5.99
StdDev	5.77	40.09
Node MTBF (day)	14.16	26.89
StdDev	15.87	25.75
Median	9	29
Node MTTR (hr)	0.87	2.15
StdDev	4.27	4.65
Median	0.15	0.28
MTTR SW	0.70	0.63
MTTR HW	100.67	7.60
MTTR M	1.15	0.55
MTTR PWR	–	4.08

Table 3.2: NCSA Platinum and Titan failure data summary.

```
 Month To Date Reliability Report for m4

                        Apr 22 2002

DATE            TYPE                       DOWN TIME
04/17/02        Software Halt                0 HOURS 10 MINS
04/17/02        Scheduled Maintenance        0 HOURS  8 MINS

There were 1 maintenance outages totalling 0 hours and 8 minutes
There were 1 software halts totalling 0 hours and 10 minutes
There were 0 hardware halts totalling 0 hours and 0 minutes
There were 0 network outages totalling 0 hours and 0 minutes
There were 0 air conditioning or power halts totalling 0 hours and
0 minutes

TOTAL MACHINE TIME IS:                     504 hours or 100 percent
TOTAL MACHINE DOWN TIME MONTH TO DATE:       0 hours or 0.00 percent
TOTAL MAINTENANCE MONTH TO DATE:             0 hours or 0.00 percent
TOTAL UNSCHEDULED HALTS MONTH TO DATE:       0 hours or 0.00 percent
TOTAL MACHINE UPTIME AVAILABLE TO USERS
    MONTH TO DATE:                         504 hours or 100.00 percent
```

Figure 3.3: Sample monthly reliability report of NCSA Origin 2000

many Myrinet M3-SPINE switch cards. The usual symptom that prompts a network card or switch card replacement is there are excessive CRC check errors. Sometimes the self-testing in a switch card may fail and lead to replacement. Cable replacements also occurred because the “ping” query packets cannot get through.

For the O2K, hardware crashes were split almost evenly between memory board failures and power supply failures. There were a few CPU board failures, but they were in the minority.

We were also told by the NCSA staff that many of the software halts occurred due to either application crashes or operating system panics. Most of the time these halts were rather short in duration because the machine would attempt to reboot itself immediately following the halt. Usually the reboot attempt was successful, so operator intervention was minimal for these halts.

The availability is lower for the O2K because when one of its machine is down, as much as one-sixth of the overall system capacity could disappear (e.g. machine B, which has 256 CPUs.) This is unlike PC clusters in which each node usually contains no more than 8 CPUs, so the availability could degrade more gracefully.

For the O2K, the machine-wise TBFs and TTRs are skewed toward small values. Eleven of the twelve machines have MTBF greater than 8 days, but the medians of TBF are mostly smaller than 4 days. For TTR, nine machines’ MTTR are greater than 2.5 hours, yet the medians are 0.3-0.9 hours. The same phenomenon also occurs on the Platinum and the Titan’s node TTR. These prompt us to study examine closely the distributions of TBF and TTR, which is the topic of the next section.

3.4 Summary

In this chapter we analyzed the failure data of three large high-performance computing systems. We found the availbility is about 98.7-99.8 percent. The Mean Time Between Failure for individual machines/nodes is 14.2-26.9 days. The Mean Time To Repair is 0.9-3.5 hours. Software halts account for the most number of outages (59-84 percent), and take the shortest time to repair (0.6-1.5 hours). Hardware problems, albeit rarer, need 6.3-100.7 hours on the average to solve.

Chapter 4

Diskless Checkpointing

The preceding two chapters show that in a distributed high-performance computing system, a variety of soft and hard errors exist and can harm application execution. To cope with the errors, the standard practice of protection in most scientific applications is checkpoint and restart.

In this chapter, we introduce an efficient method of checkpointing, called scalable diskless checkpointing. First, we give a brief overview of common checkpointing approaches in scientific computing and its associated I/O problem in §4.1. In §4.2 we describe our solution: scalable diskless checkpointing. Diskless checkpointing uses redundancy codes to recover from lost data, which we elaborate in §4.3. In §4.4-§4.7 we describe the implementation and operation details.

4.1 Checkpointing

The standard practice of fault tolerance in most scientific applications is checkpoint and restart. Checkpointing is a rollback-based recovery. A program periodically dump system state, usually values of key variables, to a persistent storage. If there is a failure, a program can restart at the previous point where they checkpointed. It avoids having a program to restart from the beginning in case of failure.

For a message-passing distributed environment, processes must coordinate to produce a consistent global system state, which consists of states of all processes and channels among the pro-

cesses. One classification of coordinated checkpointing techniques is based on whether the checkpoint is performed asynchronously or synchronously [33]. Synchronous (blocking) checkpointing is straightforward: all processes synchronize first, then dump their local state to the storage immediately. No messages are allowed to be sent bewteen synchronization and state dumping. The synchronization step, therefore, effectively flushes the channels so that only processes states need to be recorded.

Asynchronous (non-blocking) checkpointing allows processes to checkpoint at different times to reduce I/O overhead, but both the channels and processes states must be recorded. One of the asynchronous checkpoint protocols is Chandy-Lamport's distributed snapshot algorithm [34]. It assumes that message delivery is reliable and first-in first-out (FIFO). Initially, a process starts checkpointing by taking a local checkpoint and then broadcasting "marker" messages. Upon receiving the first marker, processes take a local checkpoint and rebroadcast the marker before sending any user application message. Upon receiving subsequent markers, processes record the channel state by logging the messages received since the first marker. It can be shown that Chandy-Lamport's protocol can produce a consistent global state.

Most MPI-based scientific programs only implement synchronous checkpointing because it is simpler and it fits the MPI's SPMD (single-program multiple-data) execution model perfectly. As supercomputers evolve into tens of thousands of nodes, the cost of coordinated checkpointing lies in not the synchronization part, but the huge amount of concurrent I/O from all the participating nodes.

As illustrated in Figure 1.3 in Chapter 1, the I/O subsystem in current large systems is insufficient to accomodate sudden large data flows due to synchronous checkpoint. Two possible bottlenecks exist. First, the bandwidth of bridging network between the compute farm and the storage farm is designed to serve modest I/O requests but not peak demand. Second, the disk bandwidth within the I/O farm may also be insufficient to cope peak I/O demand.

Therefore, synchronous checkpointing does not scale well beyond thousands of nodes in the current high-performance computing systems. To allow application to perform synchronous check-

pointing more efficiently, we must localize the I/O traffic. This is the basic idea of diskless checkpointing, which we elaborate in the next section.

4.2 Diskless Checkpointing

We propose an improved version of diskless checkpointing to deal with I/O problems of synchronous checkpointing on large systems. Diskless checkpointing was first proposed by Plank *et al* [35]. It is essentially a software-implemented RAID that provides a high-performance and reliable storage for intermediate or temporary data, such as checkpoint files. It replaces the traditional disk-based I/O subsystem with local memory and spare nodes to speed up the checkpointing process, enabling the user application to checkpoint more frequently.

Our version of diskless checkpointing differs from Plank's in that, in order to achieve scalability, all nodes are partitioned into equal-sized groups. Each group has one or more nodes designated as standby spares which do not involve in the computation. At checkpoint, each node saves its restart data in local memory and nodes in the same group cooperate to create redundancy codes and store them in the spares. Figure 4.1 shows the diskless checkpointing of a group of four processes and one spare.

When a compute node fails, a spare in the same group assumes the role of a compute node and the lost checkpoint is reconstructed from redundancy codes. Obviously, each group can survive node failures no more than the number of spares in the group, as shown in Figure 4.2. Another kind of catastrophic failure that can thwart diskless checkpointing is the network switch or power failure, which usually disconnect or disable a large number of nodes.

Therefore, disk-based checkpointing cannot be totally eliminated. A better checkpoint scheme would be alternate among diskless and disk-based checkpoint, depending on the performance overhead and user's need. For example, if a checkpoint method is slow, it should be performed less often. Or if there is insufficient free memory, then the checkpoint should be directed to local disks or storage farm. Also performance degradation or excessive read/write retries at a local disk usually prophesize an imminent failure, so in this case we can remove local disks from possible choices of

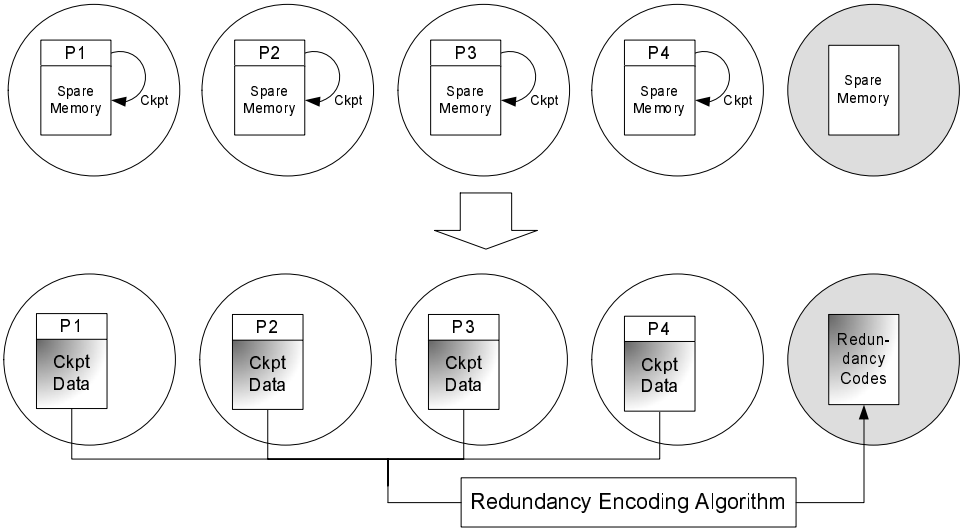


Figure 4.1: Diskless Checkpointing

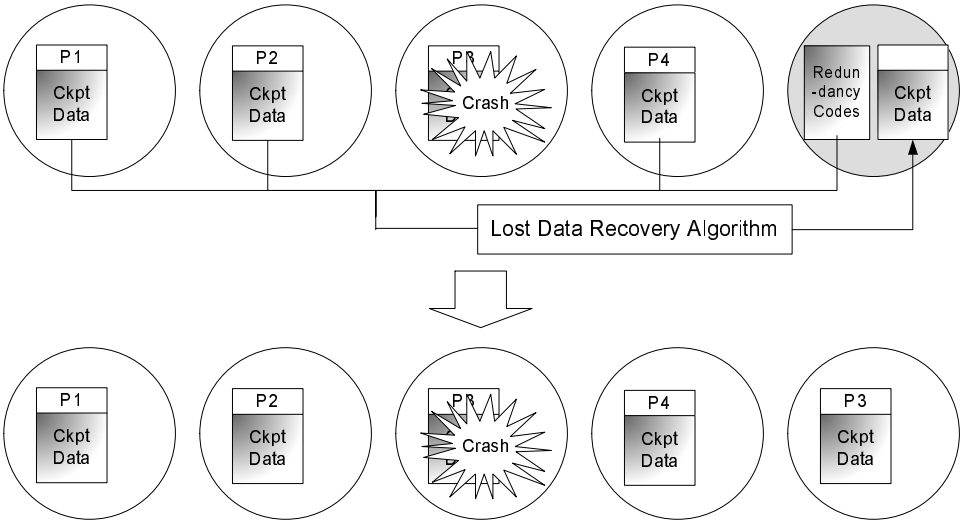


Figure 4.2: Data Recovery in Diskless Checkpointing

storage.

As mentioned above, in diskless checkpointing each process writes to in-memory storage, followed by redundancy encoding. The former is straightforward, whereas the latter will be explained in the next section.

4.3 Redundancy Encoding and Data Recovery

In diskless checkpointing, we characterize the failure using the erasure model. In this model, there is a set of data blocks, and one failure can *simultaneously* erase or alter contents of one or more blocks, and the system is able to detect which block(s) are tempered.

Under this model, we view the checkpoint data of a process as a data block. We also view two failures that occur in a short time as an erasure of two data blocks, such as a failure that occurs during the recovery for the previous failure. A recovery is to rebuild the lost data blocks from remaining data blocks and redundancy blocks.

4.3.1 The Erasure Model

To formalize the idea, assume we have a collection of n data blocks $D = \{d_1, d_2, \ldots, d_n\}$, where d_i is the checkpoint data on compute node i for $1 \leq i \leq n$. We also have a collection of k redundancy blocks $R = \{r_1, r_2, \ldots, r_k\}$, where r_i is the redundancy stored on spare node i for $1 \leq i \leq k$.

The complete, untampered dataset is $D \cup R$. If there is a failure, the erased data E is a non-empty subset of $D \cup R$, so the remaining good data is $(D \cup R) - E$. E is usually called the *error pattern* or *error symptom*. The encoding algorithm tells how to get R from D, and the corresponding data recovery algorithm tells how to rebuild E from $(D \cup R) - E$. The erased data could be either data blocks or redundancy blocks, and we do not need to recover from erased redundancy blocks.

Following the formalism, a redundancy scheme is a pair of encoding algorithm F and data recovery algorithm G such that $R \leftarrow F(D)$ and $E \leftarrow G(D \cup R - E)$. ($\leftarrow$ means "is assigned as".) Actually, F consists of k functions $F_1, F_2, \ldots, F_k$ such that $r_i \leftarrow F_i(D)$, $1 \leq i \leq k$. Therefore, $R \leftarrow F(D) = \{F_1(D), F_2(D), \ldots, F_k(D)\}$.

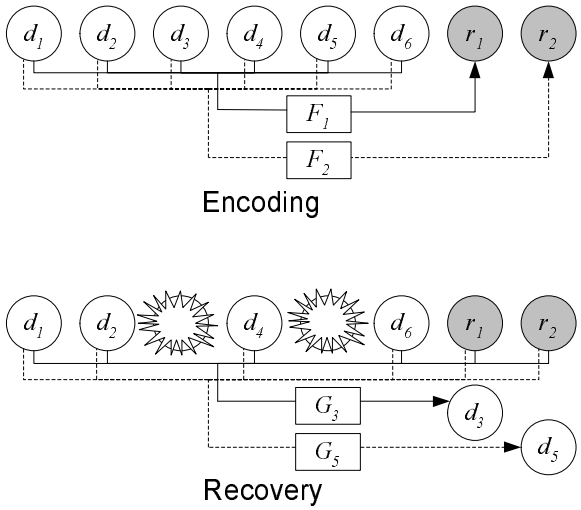


Figure 4.3: The erasure model of redundancy encoding and data recovery. There are 6 data blocks $D = \{d_1, \ldots, d_6\}$ and 2 redundancy blocks $R = \{r_1, r_2\}$. The encoding algorithm F has two encoding functions F_1 and F_2 which map D to r_1 and r_2, respectively. Suppose d_3 and d_5 are lost, then $E = \{d_3, d_5\}$. The corresponding recovery algorithms G_3 and G_5 will map $D \cup R - E = \{d_1, d_2, d_4, d_6, r_1, r_2\}$ to d_3 and d_5, respectively.

Supposedly we are only interested in recovering lost data blocks but not redundancy ones. Then the recovery algorithm G consists of n functions $G_1, G_2, \ldots, G_n$, each of which recovers the lost data block d_i by $d_i \leftarrow G_i(D \cup R - E)$, $1 \leq i \leq n$. Note that $d_i \in E$, but E can also contains other data blocks. Figure 4.3 gives an example of 6 data blocks and 2 redundancy blocks.

Depending on the redundancy scheme, G_i could be undefined for certain particular symptom E. For example, for any encoding algorithm, if $E = D \cup R$ (i.e. all nodes die), then G_i is undefined because G_i cannot recover from void. If G_i is undefined on certain E, then we say E is *unrecoverable* or *catastrophic*.

A simple application of this model is RAID 1, which is called mirroring or full replication. In RAID 1, we must have $k = n$, i.e. equally many redundancy blocks as data blocks. The encoding algorithm F is simply $F_i(D) \equiv d_i$ and thus $r_i = d_i$ for $1 \leq i \leq n$. ($\equiv$ means "is defined as.") The recovery algorithm G is $G_i(D \cup R - E) \equiv r_i$. An erasure is unrecoverable if for some i, $d_i \in E$ and $r_i \in E$, i.e. both the original data d_i and its backup r_i are lost. The main problem with the mirroring is: although the system is furnished with many spares, in the worst case two failures are

enough to stall the system.

As a result, there is a great deal of research in search for redundancy scheme that can recover from maximal possible and arbitrary error pattern E with minimal number of redundancy blocks. A good redundancy scheme should be able to recover from k arbitrary failures, provided there are k redundancy blocks. Below, we discuss two redundancy schemes on which our diskless checkpointing is based.

4.3.2 Parity Codes

In parity codes, k is fixed to be 1. The parity codes are resilient to one arbitrary failure and is the foundation of reliability of RAID 3, 4, and 5. [1] The encoding algorithm F is $F_1(D) \equiv \bigoplus D$, so $R = \{F_1(D)\}$. [2] For the recovery algorithm G, $G_i(D \cup R - E) \equiv \bigoplus(D \cup R - E)$ if $E = \{d_i\}$ and $G_i(D \cup R - E)$ is undefined if E contains more than one data block.

4.3.3 Reed-Solomon Codes

The Reed-Solomon codes are very general redundancy codes that can be found from applications from communications (satellite, cell phones, high-speed modems/DSL, digital TV) to storage (tape, hard drives, CD, DVD, barcodes). It uses k redundancy blocks and can recover from arbitrary error patterns which contain k failures or less. The mathematics behind the Reed-Solomon codes are based on an algebraic structure called finite fields (Galois fields) of power of 2, which is denoted as $GF(2^w)$. We give a sketch of its encoding and recovery processes based on the work in [36, 37].

Given n data blocks $D = \{d_1, d_2, \ldots, d_n\}$ and a supply of k spares, we use the following encoding functions to get redundancy block r_i:

$$r_i \leftarrow F_i(D) \equiv d_1 + d_2 * 2^{i-1} + d_2 * 3^{i-1} + \ldots + d_n * n^{i-1} \tag{4.1}$$

for $1 \leq i \leq k$. Putting in matrix form:

[1] The differences among level 3,4, and 5 of RAID lie in the storage locations of parity bits.

[2] Let $X = \{x_1, x_2, \ldots, x_n\}$, then $\bigoplus X = x_1 \oplus x_2 \oplus \ldots \oplus x_n$ where $\oplus$ is the bitwise XOR operation.

$$\begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_k \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{k-1} & 3^{k-1} & \cdots & n^{k-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}$$

The matrix in the middle of the equation is called the Vandermonde matrix.

Recovery from failures is more complex and has two steps: forming two failure matrices A and E and inverting a matrix.

The failure matrix A is obtained by running Gaussian elimination on a $(n+k)\times n$ Vandermonde matrix such that the upper $n \times n$ part is the identity matrix, i.e. the A has the following form:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ a_{n+1,1} & a_{n+1,2} & \cdots & & a_{n+1,n} \\ \vdots & \vdots & & & \vdots \\ a_{n+k,1} & a_{n+k,2} & \cdots & & a_{n+k,n} \end{bmatrix}$$

The other failure matrix E is simply defined as:

$$[d_1 \cdots d_n r_1 \cdots r_k]^T$$

Note that both A and E have $n + k$ rows, and each data block d_i and redundancy block r_j have a corresponding row in A and E: d_i maps to row i and r_j maps to row $n + j$ ($1 \leq i \leq n$ and $1 \leq j \leq k$)

Suppose k blocks are lost, to recover, we first delete the lost blocks' corresponding rows from both A and E. If we lose fewer than k blocks, we can randomly delete remaining rows until there are only n rows left. After removing k rows from A and E, we have $n \times n$ matrix A' and $n \times 1$ matrix E', and the data is reconstructed by

$$[d_1 \cdots d_n]^T = (A')^{-1}E'$$

The special construction of A guarantees that no matter which k rows are deleted, A' is invertible and $(A')^{-1}E'$ always gives back the data blocks.

Several aspects of Reed-Solomon codes are worth noting. First, A can be precomputed. Second, the biggest overhead in the Reed-Solomon codes is that all arithmetic operations in the above derivations are performed in $GF(2^w)$. In $GF(2^w)$, additions and subtractions are still bitwise XOR, but multiplications and divisions are non-trivial and require table look-ups. Third, the smallest size of the data blocks is w bits, where w is chosen such that $2^w > n + k$. In practice, the data blocks are decomposed into w-bit words and the above operations are performed on these w-bit words.

We adopted Plank's implementation of Reed-Solomon codes [36] and further optimize its performance as follows. Recall in formula 4.1 that each data block d_i is first multiplied by a constant. Since a block is decomposed into words, we can calculate the multiplication of every possible w-bit word with that constant beforehand and store the results in a table. During encoding the slow $GF(2^w)$ multiplication reduces to a simple table look-up. To have a reasonable table size, the value of w is chosen to be 16, which will yield a table of 2^{16} entries of 16-bit words. The precalculation step is like this

```
for (i=0; i < 65536; i++)
  RSTable[i] = GFMultiplication(i, constant);
```

Then encoding and decoding is simply a series of table look-ups that map each user data word in `source` into the result array `target`, as follows.

```
for (i=0; i < SIZE; i++)
  target[i] = RSTable[source[i]];
```

4.4 Design and Implementation

We designed and implemented the diskless checkpointing and data recovery system (DLCKPT) to experimentally verify its performance and efficacy. The DLCKPT, coupled with a fault-tolerant MPI library, LA-MPI, form a solution to the I/O and reliability problems of scientific codes running on large high-performance computing systems.

The DLCKPT infrastructure consists of two components: a lightweight user-space in-memory file system and a separate DLCKPT codec (coder/decoder) module. Figure 4.4 illustrates the overall infrastructure. The DLCKPT file system handles I/O requests from the user application and manages the memory storage, while the DLCKPT codec module performs redundancy encoding and data recovery. We designed the DLCKPT to run completely in user space, so no extra root privilege or kernel modifications are required. This greatly reduces administrative cost and gives users more flexibility.

Unlike RAID storage, which produces redundancy codes on-the-fly as clients write, in the DLCKPT the client must explicitly call a function to perform encoding. This can improve performance because most checkpoint data is huge and one bulk transfer is faster than a series of small transfers. It also simplifies the design as DLCKPT does not need to deal with the data consistency problem arised from multiple clients writing concurrently.

Below we begin with a description of the DLCKPT file system and the DLCKPT codec module in §4.5 and §4.6, respectively, followed by their interactions in §4.7.

4.5 Diskless Checkpointing In-Memory File System

Since most most applications are designed to run in the UNIX-like environment, to allow applications to use DLCKPT without too much change in the code, we have designed a pseudo file system (the DLCKPT file system) that emulates the UNIX file systems. The DLCKPT file system have two parts: the file system itself and its I/O application programming interface (API).

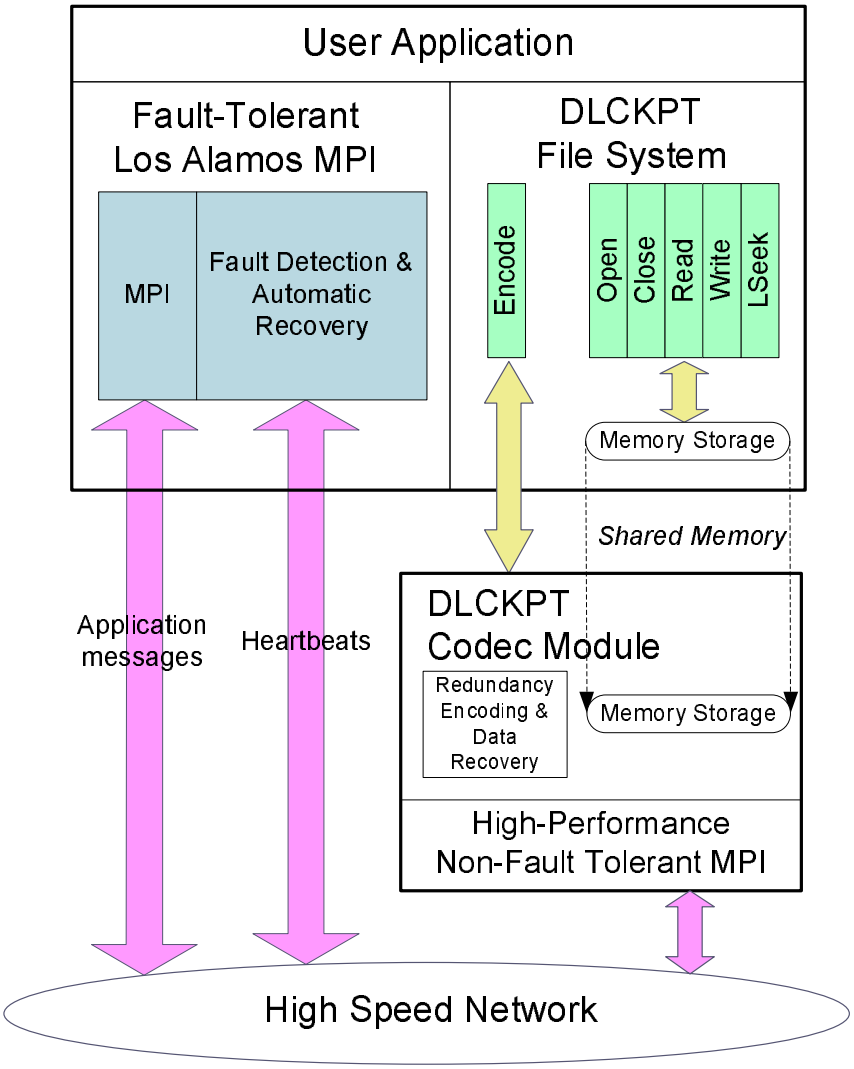


Figure 4.4: Diskless Checkpointing and Recovery System (DLCKPT) Infrastructure

4.5.1 The File System

Although the DLCKPT file system can simulate the basic operations of a UNIX file system, there are still some differences, as discussed below.

Naming

The DLCKPT file system is a local, per-process file system. It resides in the same address space as the user process, and it only handles I/O requests from the host user process. Therefore, two or more user processes on the same physical machine only see their respective DLCKPT file systems but not others' DLCKPT file systems. As such, file access rights are also unnecessary and hence are not implemented in DLCKPT.

For simplicity's sake, DLCKPT does not have directories or other sophisticated features such as symbolic links or memory-mapped files. Each file is uniquely identified by its full pathname. To simulate the semantics of UNIX file systems, when a file is created, DLCKPT will try to create it on the disk-based file system. This on-disk file is the *shadow* of the one in memory. If the shadow can be successfully created, its full pathname is obtained and is used as the file's name in the DLCKPT file system. The shadow is consulted only when file information and manipulation calls such as `fcntl`, `fstat` and `ioctl` are invoked; all I/O traffic goes to the file in memory.

File Management

We have noted that most scientific applications do not need more than a handful files in their lifetime (A checkpoint dump can be repeatedly overwritten.) Hence, we adopted a simple file management design. Unlike the UNIX file system's inode structure[3], DLCKPT file system has a file table of fixed size. The maximum number of files that can live within DLCKPT file system is constant during run-time.

[3]UNIX file systems administer files by means of inodes. Each inode corresponds to a file and contains key information such as file size and access rights. The inode approach allows the file system to grow so long as the physical storage permits.

Block Management

As with most UNIX file systems, the DLCKPT file system allocate file storage on a block basis. The allocation is dynamic, as needed. On the other hand, the block addressing of DLCKPT is different from UNIX file systems. In UNIX file systems, the block addressing is multi-level. The first few blocks of a file can be directly addressed (”direct”). If the file contains more blocks, an auxiliary block is allocated to store a list of addresses of these additional blocks (”single indirect”). If the file contains still more blocks, ”double indirect” and ”triple indirect” methods are used. We designed DLCKPT file system with performance as an aim, therefore, we only implemented the direct addressing mode. Although this limits file growth, it should not present a problem because the user should have in mind the size of checkpoint files and is free to change the limit.

We utilizied shared memory to allocate blocks. Shared memory is one of the inter-process communication mechanisms available in UNIX. A process allocates a shared memory segment using `shmget` system call. The arguments of `shmget` include the size and the access rights of the segment to be allocated. The return value is a segment ID. Unrelated processes can attach to the same segment by calling `shmat` with the same segment ID. The return value is a pointer to the segment. Once attached, accesses to shared memory are as fast as to non-shared ones because no system call or entry to kernel are required. In our design, the DLCKPT module must be able to access DLCKPT file system to produce redundancy bits, so shared memory arises as the most natural choice. Another advantage is the persistence of shared memory across process creations and terminations. Shared memory is a system-wide resource, so as long as the operating system is intact, the data in the DLCKPT file system can be retrieved even if the host process crashes.

A potential risk of using shared memory is the wild writes to the shared memory region due to application bugs or transient errors. Adding CRC (cylic redundancy check) codes is one possible protection to the data from being tampered. In our current implementation, we assume the integrity of checkpoint in the memory storage is immaculate, just like the data on the disk.

4.5.2 I/O Application Programming Interface

The DLCKPT file system is implemented as a user library. It provides two sets of I/O API: the traditional UNIX low-level I/O system call interface (e.g. `open`, `read`, `write`), and the DLCKPT-prefixed I/O interface (e.g. `dlckpt_open`, `dlckpt_read`, `dlckpt_write`.) The former is implemented as a wrapper of the latter, so they share the same functionality and semantics.

UNIX I/O Call Interception

However, only implementing the UNIX I/O system call interface is not sufficient. Many applications perform I/O without calling UNIX I/O interface directly. For example, C language provides the buffered I/O such as `fopen`, `fread`, and `fprintf`, C++ has stream I/O such as `ifstream` and `ostream`, and Fortran has formatted I/O. Furthermore, some applications use special I/O libraries to access files in certain format such as HDF[38] and XML[39]. Nevertheless, the common denominator at the heart of these I/O libraries is the UNIX I/O interface.

To intercept UNIX I/O calls emitted from an application, we have considered several methods proposed by Thain and Livny in [40]. The methods are either at kernel level or user level. The kernel-level solutions are usually through kernel patches or kernel modules. They can add administrative burden and cause system instability. The weaknesses of kernel-level solutions are the strengths of user-level solutions: the file system acts like an sandbox isolated from the system kernel and the user has total control over it without security or protection concerns.

Therefore, we decided to adopt a user-level solution: the static linking technique, which forces all I/O libraries linked to the application to call our UNIX I/O interface. Generally, the user application is linked *dynamically* to I/O libraries to reduce sizes of binaries, so the I/O libraries code is not part of the application binaries. As such, UNIX I/O calls emitted by I/O libraries do not go through our interface. To overcome this problem, we use linker controls to force *static* linking of the user application to I/O libraries. This creates a stand-alone application binary that can self-serve all UNIX I/O requests. We have tested and validated this approach on both Linux version 2.4 and IBM AIX version 5.x.

Storage Control

The DLCKPT file system provides a global variable `dlckpt_enabled` for users to control the storage preference when a file is opened. If `dlckpt_enabled` is true, this file will be stored in the DLCKPT file system; otherwise, the file will be handled as if DLCKPT is absent. The following code shows how the DLCKPT interface can be used in an application. In this example, both files are created and written using the buffered I/O routines of the C language. The file referred by `f1` is in the memory and file referred by `f2` is on the disk.

```
FILE *f1, *f2;

dlckpt_enabled = 1;
f1 = fopen("file1", "w");
dlckpt_enabled = 0;
f2 = fopen("file2", "w");

.
.
fprintf(f1, "Hello World");
fprintf(f2, "Hello World");
```

The above example presents a problem: how can DLCKPT tell the `fprintf` call on `f1` should be handled by DLCKPT file system and the same call on `f2` should be relegated to the operating system? After all, we did not modify the `fprintf` routine in C library.

We mentioned that C library, among most I/O libraries, also makes use of UNIX I/O calls to satisfy I/O requests. To a UNIX operating system, all open files are referred to by file descriptors (non-negative numbers), and every I/O call requires the file descriptor as one of arguments. Therefore, we solved the above problem by using disparate ranges for file descriptors. First, we use `getrlimit` system call to get the upper limit of file descriptors that can be assigned by the operating system. For example, this limit is 1024 on most Linux systems. Then for any file that will reside in memory, the DLCKPT file system returns a file descriptor greater than 1024, say 1025, to it. Thus, by only examining the file descriptor's value, the DLCKPT is able to recognize the file's storage location and handle I/O requests accordingly.

4.6 Diskless Checkpointing Codec Module

The DLCKPT codec module takes the DLCKPT file system contents and produces XOR parity or Reed-Solomon redundancy codes. In case of failure, the DLCKPT codec module also reconstructs lost data from redundancy codes, as described in §4.3.

We implemented the DLCKPT codec module as an MPI program. There are two advantages of doing this. A crucial step in the diskless checkpointing is to send local encoding results across the network. An MPI program enables us to harness fast interconnections without low-level details of network hardware programming because MPI implementations tailored for most interconnects are widely available. Another strength is to reduce programming complexity because the DLCKPT codec module uses a great deal of collective communications which are readily implemented as MPI calls.

We designed the DLCKPT codec module as an independent program instead of a user library. The reason is as follows. The user application is made fault tolerant by being linked to the LA-MPI library. For compatibility reasons, the LA-MPI library only supports UDP as the message transmission mode. If the codec functionality is part of DLCKPT file system, the codec will be forced to use LA-MPI library, hence slow down checkpointing performance. To strike a balance between performance and fault tolerance, we decided to isolate DLCKPT codec module from DLCKPT file system, so it can use the system-dependent high-performance MPI library.

4.7 Diskless Checkpointing in Operation

In this section, we will discuss how the DLCKPT file system and the DLCKPT codec module cooperate to produce a diskless checkpoint. We will cover the data recovery part together with the fault-tolerant LA-MPI library in the next chapter.

Roughly speaking, during run-time, the DLCKPT codec module acts as a server which takes commands from the DLCKPT file system. The DLCKPT file system in the root process sends commands to the root process of DLCKPT codec module. Non-root processes of DLCKPT codec

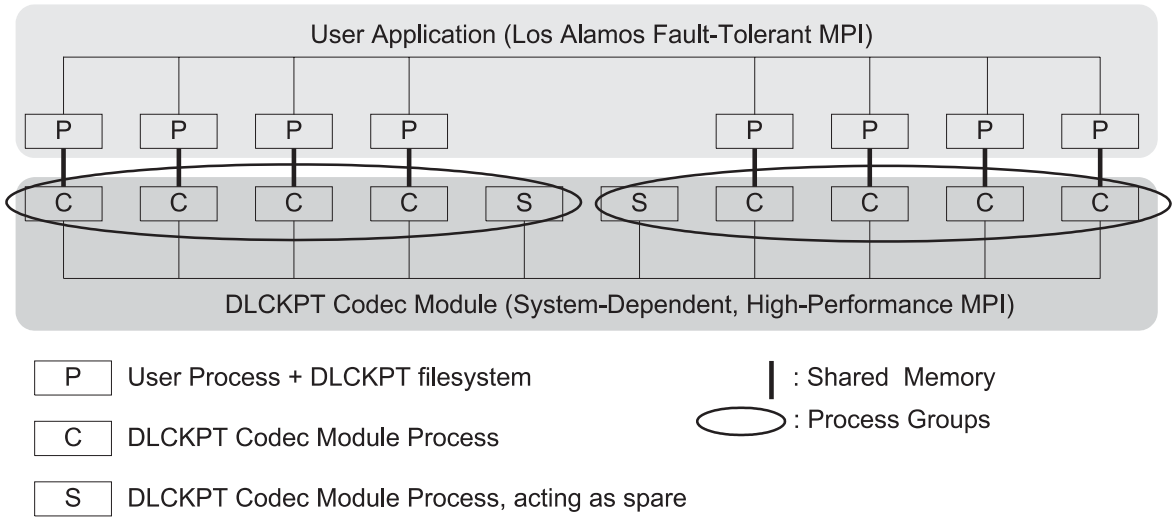


Figure 4.5: Diskless Codec Module during Run-time

module communicate implicitly with their corresponding DLCKPT file system via shared memory. The relationship is illustrated in Figure 4.5. Their run-time interactions are described in the following steps. The steps 1-3 are done once during the initialization, and the steps 4-5 are repeated for every diskless checkpoint.

1. The DLCKPT codec module is launched and number of groups and operation mode (regular or recovery) are specified in the command-line arguments. Its root process sets up a TCP port to listen to commands from the DLCKPT file system of the root process of the user application. It also writes the host name and port number information to a file in the user home directory.

2. The user application is launched and it should have less number of processes than the DLCKPT codec module does. The user application proceeds to do computations. At checkpoint time, every user process writes to the DLCKPT file system and then calls the `dlckpt_encode` function. In this function, the DLCKPT file system gathers user processes information (shared memory segment IDs and host names) at the root process. The root process reads the file saved in the previous step and makes contact with the DLCKPT codec module. It sends user

processes information to DLCKPT codec module.

3. Upon receiving user processes information, the DLCKPT codec module matches user processes with its own processes by host names. Some of the DLCKPT codec module's processes do not match any user processes, so they act as spare process, as shown in Figure 4.5.

   The DLCKPT codec module partitions its processes into groups. The number of groups is specified in Step 1. The partition is conveniently done by calling `MPI_Comm_split` function. This function creates new communicators[4] from an existing communicator. It takes an integer-value argument "color." Processes with the same "color" are placed in the same new communicator. In our case, we assign different colors to process groups and create `group_comm` from the default global communicator as follows:

```
MPI_Comm_split(MPI_COMM_WORLD, group_color, 0, &group_comm);
```

4. The root process of the user application sends ENCODE command to the DLCKPT codec module and waits for response.

5. Upon receiving ENCODE command, the DLCKPT codec module starts encoding. Non-spare processes attach to user processes' shared memory segments using the information collected in step 3. If a group has more than one spare process, the encoding algorithm to be used is Reed-Solomon codes. In this case, each non-spare process performs the local encoding first and then merges the result. Otherwise, the merge is applied directly. We implemented the code as the following:

```
if (I am spare) {
  in_buf = blank_buf
  allocate result_buf
}
else {
  attach to user_data shared memory segment;
```

[4] In MPI, a process group is called "communicator". A global communicator named `MPI_COMM_WORLD` always exists throughout the application's run-time. A process can belong to multiple communicators.

```
  if (Reed-Solomon codes)
     in_buf = local_encode(user_data);
  else
     in_buf = user_data;
}

MPI_Reduce(in_buf, result_buf, data size, MPI_CHAR, MPI_BXOR,
           spare process's rank, group_comm);
```

The `MPI_Reduce` call applies bitwise XOR (specified by `MPI_BXOR`) over `in_bufs` supplied by all processes and places the outcome in a spare process' `result_buf`. The reduction is performed on a group basis, as specified by `group_comm`.

When all groups are done, the root process of DLCKPT codec module sends DONE reply to the user application, completing one diskless checkpoint.

4.8 Summary

In this chapter we first mentioned the I/O bottleneck faced by paralell applications running on large systems. We then proposed the scalable diskless checkpointing to solve this problem. We introduced the concept of diskless checkpointing, which is basically a software-implemented RAID that uses memory as storage to provides a fast and reliable storage for temporary data, such as checkpoint files. It also uses redundancy codes (parity codes or Reed-Solomon codes) to enable recovery from loss of data.

We have also implemented the diskless checkpointing. The framework (DLCKPT) contains a user-space in-memory file system and a codec (coding/decoding) module. The DLCKPT file system is realized as a user library, whereas the DLCKPT codec module is written as an MPI program. Together with the fault-tolerant LA-MPI library, which is the topic of the next chapter, they form a complete infrastructure of high-performance checkpointing and recovery for parallel applications.

Chapter 5

A Fault Tolerant MPI Library

In the preceding chapters, we have discussed the design and implementation of a scalable diskless checkpointing (DLCKPT) infrastructure. The DLCKPT, coupled with a fault-tolerant communication library form a solution to the I/O and reliability problems of scientific codes running on large high-performance computing systems.

The theme of this chapter is a fault-tolerant communication library, which is responsible for recovering from failures (process crashes.) Since MPI (Message Passing Interface) is widely used in the high-performance computing community, we chose to enhance an existing MPI library, LA-MPI [32], by adding automatic recovery and scalable heartbeat monitoring functions required by the DLCKPT infrastructure.

5.1 Reliability Issues in MPI

MPI is a specification for the message-passing SPMD (Single-Program, Multiple-Data) parallel programming paradigm. After over ten years of development and evolution, MPI is now widely adopted as a standard for parallel programming, and its implementations can be found on systems ranging from PC clusters to supercomputers. Most MPI libraries conform to the MPI 1.1 standard [3], which provides little support for reliability.

MPI 1.1 has two pre-defined error handlers: `MPI_ERRORS_ARE_FATAL`, which is the default

one and causes the user application to abort all executing processes, and `MPI_ERRORS_RETURN`, which returns an error code to the user. The user can register customized error handlers by `MPI_Errhandler_create()` and `MPI_Errhandler_set()` calls.

However, it is also clearly stated in the MPI 1.1 specification that " *after an error is detected, the state of MPI is undefined... A user-defined error handler, or `MPI_ERRORS_RETURN` does not necessarily allow the user to continue to use MPI after an error is detected... An MPI implementation is free to allow MPI to continue after an error but is not required to do so.*"

In reality, we found that in most MPI 1.1 implementations, the user-defined error handler is triggered only when incorrect arguments are passed to the MPI calls, such as a non-existing destination in a send operation.

For catastrophic errors such as the abnormal termination of processes during run-time, most MPI 1.1 implementations simply abort the execution, even if the user has registered an error handler. Furthermore, there is no MPI function that allows the user to create new processes during run-time because MPI 1.1 adopts a static group membership model. The group membership is determined at the time the user application is started, and the group is named `MPI_WORLD_COMM`. Any loss of a MPI process effectively creates a "hole" in `MPI_WORLD_COMM` that cannot be mended in any way.

5.2 Introduction to the LA-MPI

Developed by the Los Alamos National Laboratory, the Los Alamos MPI Library [32] is an end-to-end network fault-tolerant message passing system for tera-scale clusters. We chose it as our baseline MPI library for further improvement because it already features several reliability functions, such as fail-over across multiple interconnect networks, message checksum, and heartbeat monitoring. On the other hand, it[1] still lacks the ability to recover from process failures and a scalable protocol of heartbeat monitoring. Our goal is to add these functions to make it work seamlessly with the DLCKPT system developed earlier.

Before describing our reliability enhancement, we would like to give a brief introduction of LA-

[1] LA-MPI library version 1.1.4

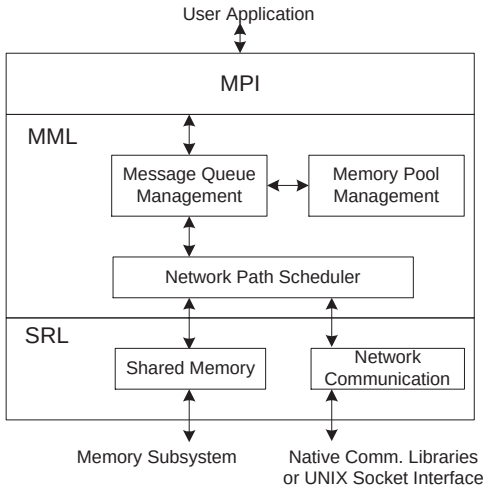


Figure 5.1: LA-MPI communication core.

MPI's software architecture and its run-time operations. More details on the LA-MPI architecture can be found in [32].

5.2.1 Software Architecture

The communication core of the LA-MPI has a three-layer architecture shown in Figure 5.1.

- The MPI layer glues the user application and LA-MPI. It contains implementations of collective communication, which are based on point-to-point transport primitives of the next layer. It also implements assorted MPI management and control functions such as `MPI_Comm_rank`.

- MML is the Memory and Message Layer. For memory part, LA-MPI has its own memory management instead of using `malloc`/`free`.

  For messages, LA-MPI keeps several queues and binds messages to different network paths. The queues record requests posted by non-blocking MPI calls such as `MPI_Irecv()`, and blocking MPI calls such as `MPI_Recv()` are implemented on the non-blocking counterparts.

  Initially, a call to `MPI_Irecv()` will put a request in either *PostedWildcardRecv* or *PostedSpecificRecv* queue, depending on whether the message tag is a wildcard or not. If an

incoming message fragment matches with any pending request in the above two queues, this pending request moves to and stays in *MatchedRecv* until the whole message is received.

A network path implements the TCP-like reliable transport layer of point-to-point communication over a particular communication substrate, may it be a high-speed interconnect, an existing protocol (e.g., UDP) or shared memory. Multiple paths may co-exist between a pair of source and destination in multi-rail networks. The job of the Network Path Scheduler is to control how messages are tied to the network paths. For example, it may stripe one message over two paths or assign messages to different paths in a round-robin style to improve network utilization. It can also support fail-over of multi-rail networks.

- In SRL (Send and Receive Layer), outgoing messages are broken into fragments whose size fits the underlying network paths, and incoming fragments are queued and re-assembled.

  As in TCP/IP, each fragment is attached with a unique sequence number in case the fragment is lost or tampered and needs re-transmission. Outgoing message fragments are first attached to *FragsToSend* list, and once sent, are moved to *FragsToACK* list. Any fragment staying too long on *FragsToACK* is retransmitted. An incoming uncorrupted fragment that matches a request in the request queue is put into either *OkToMatchRecvFrags* or *OkToMatchSMPRecvFrags* queue, depending on whether it is off-host or on-host traffic. Unexpected or out-of-order incoming fragments are stored in *AheadOfSeqRecvFrags* queue.

The above communication core is linked to the user application and becomes part of a user process. In addition, LA-MPI also has a master-daemon code and a client daemon code that execute on their own. Their functionality is best discussed in the context of the run-time operations of LA-MPI, as below.

5.2.2 Run-time Operations

The run-time process relationship of LA-MPI is shown in Figure 5.2. There are three kinds of processes: a master daemon, client daemons, and user processes. The master daemon, `mpirun`,

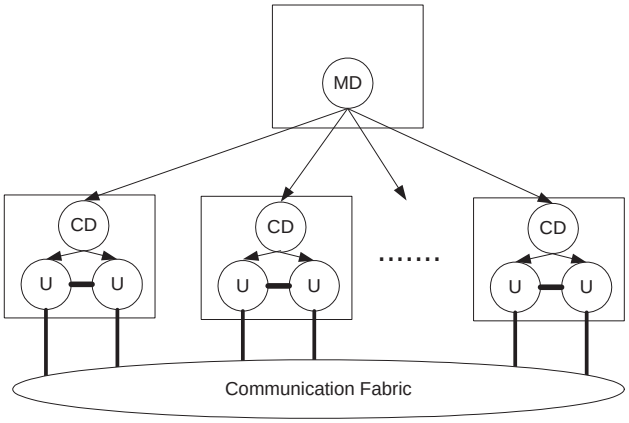


Figure 5.2: LA-MPI run-time process relationship. Each box denotes a physical node, and each circle denote a process. MD is master daemon, CD is client daemon, and U is user process. An arrowed line from A to B means A checks B's health periodically. The thick solid line is MPI message communication among user processes, which can be intra-node (via network) or inter-node (via shared memory). Each node can have only one client daemon but can accommodate many user processes.

is responsible for spawning client daemons on remote nodes, checking health of client daemons periodically, and handling console I/O (e.g., collecting remote processes' output to the `stdout` device.)

Each node runs one client daemon, which spawns one or more user processes and then enters the daemon mode. In this mode, the client daemon constantly checks for control messages such as heartbeat from the master daemon and monitors the status of user process via `SIGCHLD` signal and `waitpid()`. The user processes run the user application and perform MPI communications with peer processes.

5.3 Reliability Enhancement

Our enhancement to the LA-MPI is to add automatic recovery capability and a scalable heartbeat monitoring scheme.

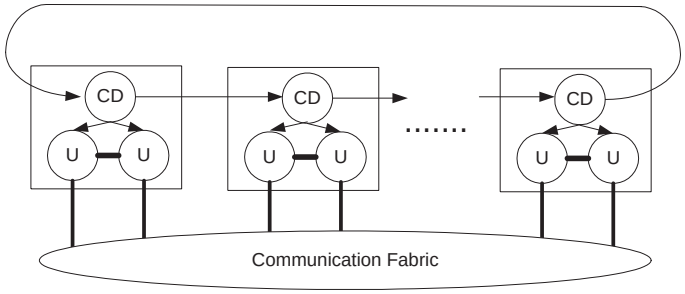


Figure 5.3: Scalable heartbeat monitoring.

5.3.1 Scalable Heartbeat Monitoring

Heartbeat monitoring in the current LA-MPI (version 1.1.4) library is a centralized scheme in which a master daemon oversees all other daemon processes, as in Figure 5.2. Our new scalable scheme gets rid the master daemon. Instead, it creates a logical ring among all client daemons. Each client daemon heartbeats its downstream neighbor in the ring and can trigger a recovery if its downstream neighbor failed, as shown in Figure 5.3.

It should be noted that the heartbeat monitoring only checks the responsiveness of peer client daemons. It does not check whether the user process is making progress or has been frozen due to failures, because the MPI library is separated from user application logic.

5.3.2 Automatic Recovery

The system or the user should supply a list of spares at the launch of an application. When the heartbeat monitor detects a failure, the LA-MPI removes one spare from the list and restarts the failed processes on the spare. All of these recovery operations are *semi-transparent* to the user process in the sense that the user application has to be slightly modified to cooperate with LA-MPI's automatic restartability.

The user should write a *recovery callback function* which cleans prior dynamically allocated memory objects, loads the last checkpoint, and re-initializes the process state. As most scientific applications are already able to resume execution from checkpoint, the additional programming ef-

fort is minimal. The following code shows how the user application is modified to use the automatic recovery feature.

```
extern jmp_buf LAMPI_Recovery;

if (setjmp(LAMPI_Recovery) == 0)
  MPI_Init(&argc, &argv);
else
  recovery_callback_function();
```

The application first registers a long jump buffer `LAMPI_Recovery`. using `setjmp()`. The first time the `setjmp(LAMPI_Recovery)` is executed, it returns 0. When there is a recovery, LA-MPI will call `longjmp(LAMPI_Recovery)`, after which the control flow jumps to `setjmp(LAMPI_Recovery)` and returns a non-zero value, hence initiates the user-provided `recovery_callback_function()`. The usage of `setjmp` and `longjmp` can be found in most UNIX programming manuals.

Although in effect the automatic recovery is like using a shell script to automates the restart of the user application, the advantage of our approach over user-directed restart is the small overhead compared to that of a full-scale re-launch of the user application, which can take long on thousands of nodes [41].

5.4 Implementation

Our modification to the LA-MPI code base is mostly inside the the commmunication core and the client daemon code. Here we discuss details of our implementation.

5.4.1 Scalable Heartbeat Monitoring

Each client daemon in the heartbeat ring sends a heartbeat message to its downstream peer every second. If any client daemon fails to receives a heartbeat within a fixed period, it initiates a recovery. If any member detects an abnormal termination of any of its user processes (via `SIGCHLD` signal and `waitpid()`), it suicides and will later be detected by absence of heartbeat.

5.4.2 Automatic Recovery

After the client daemon detects the failure of its peer, it initiate the recovery. We need to deal with both LA-MPI itself and the user application. First, the LA-MPI commmunication core restarts itself by resetting its internal data structures. Then the LA-MPI uses `longjmp()` to invoke the user-provided recovery callback function as mentioned earlier to complete the user application recovery.

The whole LA-MPI recovery follows the protocol below. Note that the current version of recovery protcol cannot handle multiple failures (e.g. due to network switch failure) or failures of two adjacent nodes in the heartbeat ring.

Let A denote the client daemon that detects a failure, then A is the coordinator of recovery.

1. A checks the groupwise spare list. If there is still one available, it broadcasts a `MPIFT_RECOVERING` message and tries to spawn a new client daemon on the spare. Otherwise, the execution cannot continue and must halt.

2. A notifies its own user processes by sending `SIGUSR2` signals to each of user processes.

3. On receipt of `MPIFT_RECOVERING`, all functioning client daemons pause the execution of their user processes and wait for instruction from A.

4. A spawns a new client daemon on a spare. If the new client daemon fails to start, it is regarded as multiple failures and the execution will halt.

5. The notified user processes jump to `SIGUSR2` signal handler function, which is inside the LA-MPI library code.

   In this handler functions, data structures in the communication core of LA-MPI are reseted. Then the user processes suspend and wait for further instruction from their client daemons.

6. When A successfully spawns a new client daemon, it disseminates the updated network and node information to other client daemons. All client daemons then update their information, reanimate the user processes to continue, and resume to normal mode of operation.

7. The user processes jump back to the user application code from inside the LA-MPI library using `longjmp()` and execute the recovery callback function (e.g. to load the checkpoint) to complete the recovery.

Figure 5.4 and 5.5 show the finite-state diagrams of this protocol.

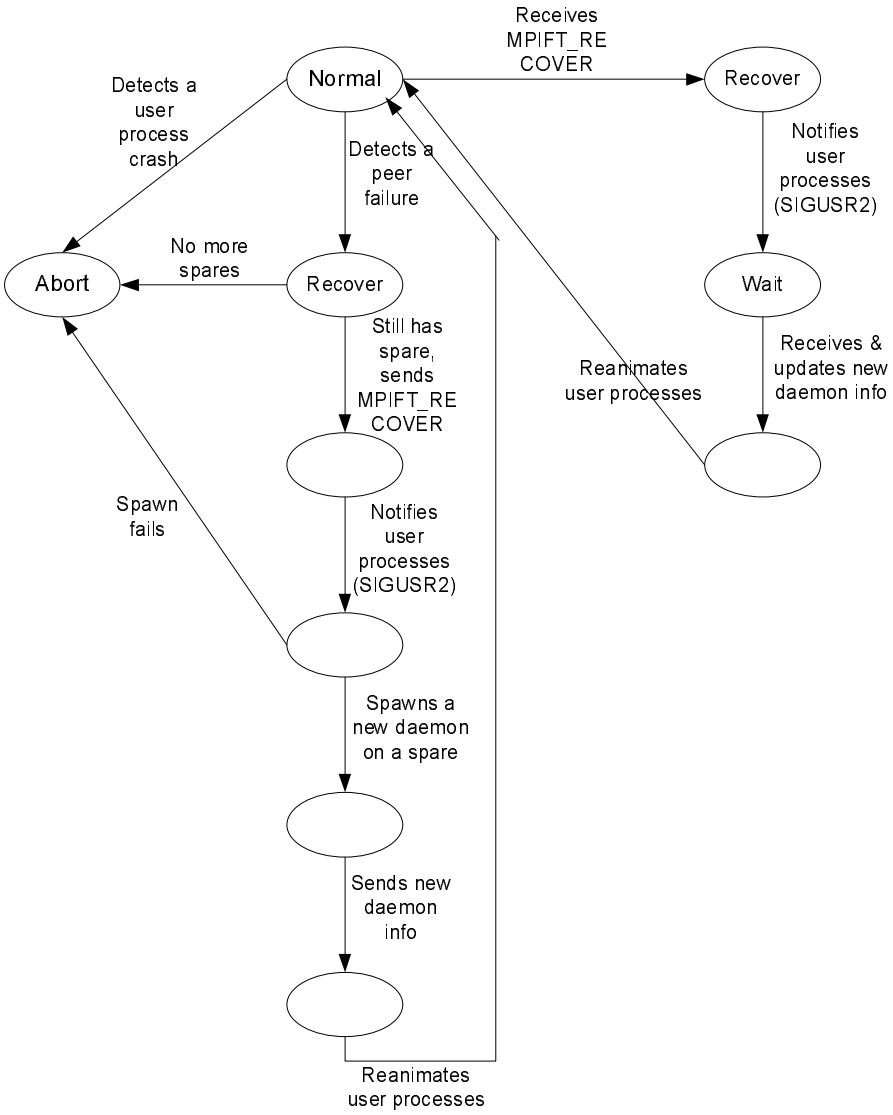


Figure 5.4: LA-MPI recovery protocol: Client daemon.

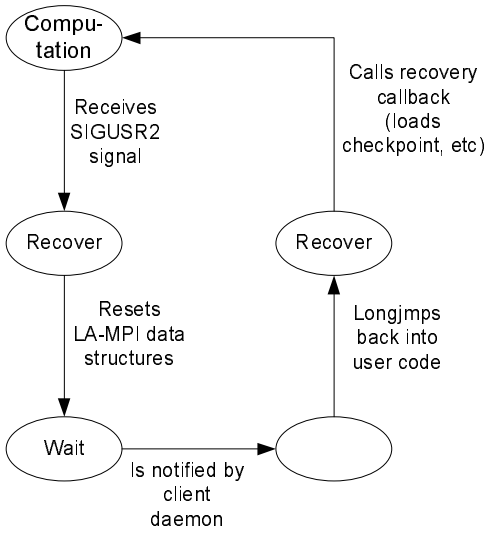


Figure 5.5: LA-MPI recovery protocol: User process.

5.5 Recovery of the DLCKPT System

The above recovery protocol works independently of the DLCKPT system. As we mentioned in §4.6, the DLCKPT codec module is also an MPI program but linked to a non-fault-tolerant MPI library for high performance. Hence, a node failure can also crash the DLCKPT codec module and render the DLCKPT system crippled.

To integrate the DLCKPT code module into the LA-MPI recovery protocol, the following two steps of the LA-MPI recovery protocol is modified:

1. A checks the spare list. If there is still one available, it broadcasts a `MPIFT_RECOVERING` message and tries to spawn a new client daemon on the spare. Otherwise, the execution cannot continue and must halt.

   A also launches the DLCKPT codec module in the recovery mode (can be specified by the command-line option) and supplies it with spare and node information.

4. The DLCKPT codec module determines whether the checkpoint data can be recovered. If all spares in the group that contains the failure are used up, the recovery cannot proceed. In this

situation, the DLCKPT notifies the LA-MPI to abort.

Otherwise, the DLCKPT codec module reconstructs the lost checkpoint on a spare. The information about this particular spare is sent back to A.

A spawns a new client daemon on this spare. If the new client daemon fails to start, it is regarded as multiple failures and the execution will halt.

5.6 Discussion

The main overhead of the automatic recovery lies in broadcasting the node failure notification, restarting the failed processes, and distributing the new node's address and port information. In particular, the newly-spawned client daemon must receive a table of network addresses of all other processes. The size of this table grows in proportion to the node count. In practice, a process may not need the complete network address information because it probably only communicates with a small set of peer processes. So an improvement would be storing the complete network address table on a subsets of nodes ("name servers") only, and other nodes can query these name servers for network address of peer processes.

As a proof-of-concept and as a complementary part to diskless checkpointing, our current scalable heartbeat monitoring and recovery protocol are not sophisticated enough to recover from failures of two or more adjacent processes. We also assume node failures are "hard," thus every failure needs a spare to resume the execution. This limitation can be relaxed by attempting to restart the process on the original node and then a spare node.

5.7 Summary

We introduced a fault-tolerant MPI library, LA-MPI, and discussed our improvement to it to enable automatic recovery. The enhanced LA-MPI uses a scalable heartbeat monitoring to detect abnormal process crashes. Following a crash, the enhanced LA-MPI can automatically respawn the failed process on a spare. It also works seamlessly with the DLCKPT system to reconstruct the lost

checkpoint data on a spare. The user application code only needs small modifications to utilize this automatic recovery feature.

Chapter 6

Experimental Results

We have conducted a series of experiments to assess the performance of the DLCKPT system on large systems. We have used an I/O benchmark and two real scietific applications on large clusters and measured the performance.

6.1 Experimental Environment

We used three Linux-based PC clusters in our experiments. Their hardware and software are described below.

1. NCSA TeraGrid TeraGrid is a multi-institution long-term collaboration to deploy high-performance computing resources in a geographically distributed environment for the science community. In addition to providing an aggregate of 20 teraflops of computing power, TeraGrid also hosts petabytes of storage, high-resolution visualization devices, and grid computing toolkits.

   The TeraGrid system at the National Center for Supercomputing Applications (NCSA) site is a cluster of 889 nodes. All nodes are based on Intel Itanium 2 processors. There are two access nodes, each of which has four processors and 8 GB ECC memory. The compute nodes were deployed in two phases. There are 256 phase-one nodes, each of which has dual 1.3 GHz processors and 4-12 GB ECC memory. There are also 631 phase-two nodes, each of

which has dual 1.5 GHz processors and 4 GB ECC memory. The workload is managed by the Portable Batch System. The peak performance is 10 teraflops.

There are three interconnect fabrics: Myrinet 2000 [19] serves inter-node network messages, and Gigabit Ethernet and Fiber Channel serve I/O traffic.

2. SDSC TeraGrid The TeraGrid system at the San Diego Supercomputer Center has a smaller but similar hardware and software configuration as the one at NCSA. It has 384 dual 1.5 GHz Itanium 2 nodes and a peak performance of 4 teraflops.

3. NCSA Tungsten The NCSA Tungsten is a PC cluster consisting of 1,450 Dell PowerEdge-1750 nodes. Each node has dual 3.2 GHz Pentium 4 Xeon processors and 3GB of ECC memory. The nodes are connected by the Myrinet 2000 high-performance interconnect. The peak performance achievable on the 1,250 nodes is 15 teraflops.

   In practice, Tungsten is divided into five identical compute sub-clusters, one test/debug sub-cluster, and a set of I/O servers. Each compute sub-cluster has 256 compute nodes, one interactive access node, and two management nodes. The test sub-cluster has 64 compute nodes plus administration and login nodes.

6.1.1 Interconnect

In the preceding chapters, we have stressed the importance of network speed on the performance of diskless checkpointing, so here we give more details of the interconnect used by the systems on which we experimented.

All of the three clusters use Myrinet 2000 high-speed interconnect to serve inter-node traffic. Each node is equipped with a single-port Myrinet M3F-PCIXD-2 network card. This card uses 64-bit 133 MHz PCI-X bus interface which has a limiting throughput of $8 \times 133 = 1064$ MB/s. Each card contains a Lanai-XP RISC processor operating at 225 MHz and 2 MB of local memory. With fiber optic cables as transmission medium, the sustained data rate (for large messages) measured at application-level is 248 MB/s in uni-directional mode and 489 MB/s bi-directional. The latency for

short messages (< 100 bytes) is 6.3 microseconds.

6.1.2 System Software and Performance Fine-Tuning

All of the three clusters run Linux version 2.4. The compiler used to compile all benchmark and application codes is Intel C Compiler version 8. The communication middleware used on TeraGrid is MPICH-GM library version 1.2.5, which is based on MPICH [24] and Myrinet GM low-level message-passing library. NCSA Tungsten employs a proprietary MPI library called ChaMPIon/Pro from the MPI Software Technology, Inc. The version is 1.1.1.

We also made the following arrangements to optimize the network performance in our experiments.

Message Receive Mode

The first enhancement is in the communication middleware. MPICH-GM supports three modes of message receives: polling, blocking, or hybrid. By default, MPICH-GM uses polling which has the most favorable performance for short messages. However, we found the hybrid mode, in which MPICH-GM automatically enters blocking mode after one millisecond of failed polls, yielded at least four times better performance than the polling mode. The reason is in practice, the checkpointing data is large, so DLCKPT module uses very long messages. The busy waiting in the polling mode is not only unhelpful but can harm performance for bulk transfers. The hybrid mode is enabled by supplying a `--gm-recv hybrid` switch to `mpirun`.

The ChaMPIon/Pro on NCSA Tungsten adopts a message receive mode very similar to the hybrid mode of MPICH-GM, so we did not use any additional tuning.

Node Placement

The second enhancement is node placement. Ideally, all pairs of nodes can be assumed equidistant, so the message transfer time between any pair of nodes is constant. For large clusters this assumption is not true: the communication between two nearest neighbors can achieve twice more

bandwidth than that of two nodes far apart [42]. Although in diskless checkpointing we partition nodes into groups, this does not mean nodes of a group are *physically* close. An obvious reason is that we have no control over the nodes the batch scheduler assign to our jobs. Another reason is that even if the assigned pool of nodes have good affinity, without careful coordination we will not be able to take its advantage. Therefore, throughout the experiments, we have manipulated the node list fed to `mpirun`, the launcher of MPI applications, to enforce node affinity.

6.2 Checkpointing Performance

We wrote a simple micro-benchmark to assess diskless checkpointing performance under various configurations. A configuration is defined by three parameters: m, n, and the number of groups. Group-wise, m is number of clients and n is number of spares. For convenience we denote m:n. In a run, each client dumps 200 MB of data to DLCKPT file system and performs redundancy encoding. The time to accomplish this task is recorded and the dumping process is repeated until the job has exhausted its allocated time. Usually 15-50 such measurements will be taken and the average is reported. The aggregate checkpointing rate is calculated accordingly. Because all of the large systems we used impose the same cap of 256 nodes on job size, we only tested up to 512 clients in the experiments.

The results of 10:1, 20:1, 40:1, 60:1, 80:1, and 100:1 (one spare per group) configurations are presented in Figure 6.2 and 6.3.

We also experimented configurations in which two or three m:1 groups are merged into a larger group, so each group has two or three spares and runs the Reed-Solomon codes. The concept is shown in Figure 6.1.

The results of 20:2, 40:2, 80:2, 120:2, and 160:2 are illustrated in Figure 6.4 and 6.5, and 30:3, 60:3 and 120:3 in Figure 6.6.

The results clearly show excellent scalability: the checkpointing time increases only a little when more clients were added. In one spare per group scheme, it is possible to obtain 9-12 GB/s of throughput. In two or three spares per group scheme, the performance is slowed down by a

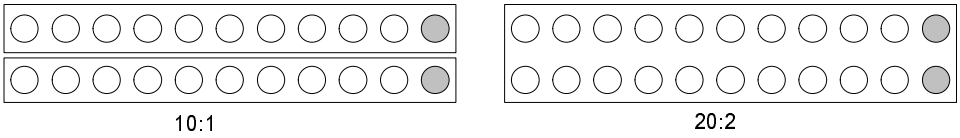
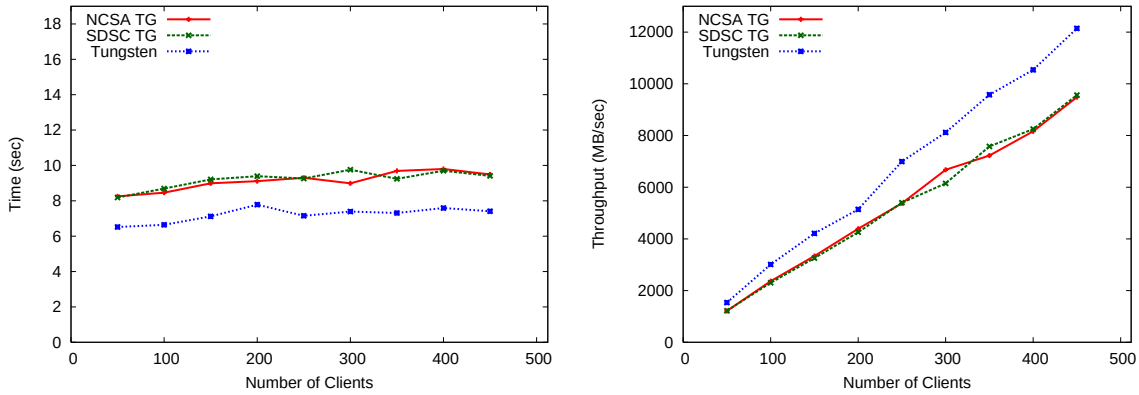
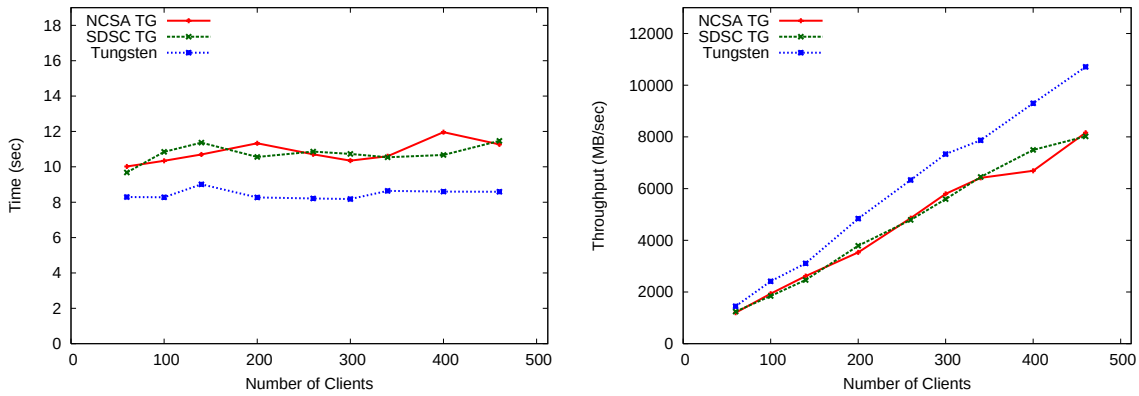


Figure 6.1: Different diskless checkpointing configurations. The circles denote nodes and gray ones are spares. The rectangles denote groups. Two 10:1 groups can be merged into one 20:2 group.

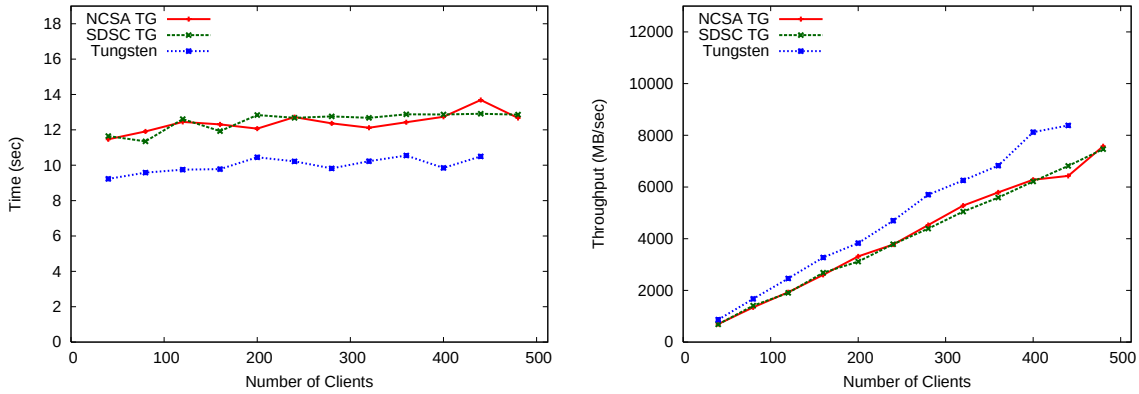
factor of 2-2.5 and 3.2-4.5, respectively, as can be seen in Figure 6.8. This is expected because first, Reed-Solomon algorithm makes n reduction passes over the data where n is the number of spares per group. Second, an additional encoding procedure is required before each reduction. Third, the group size is larger, so each reduction takes more time. This increase of time along with group size is logarithmic. Figure 6.7 validates this claim. However, the logarithmic growth depends on the MPI library being used. More details will be discussed in the next chapter.



10:1 configuration

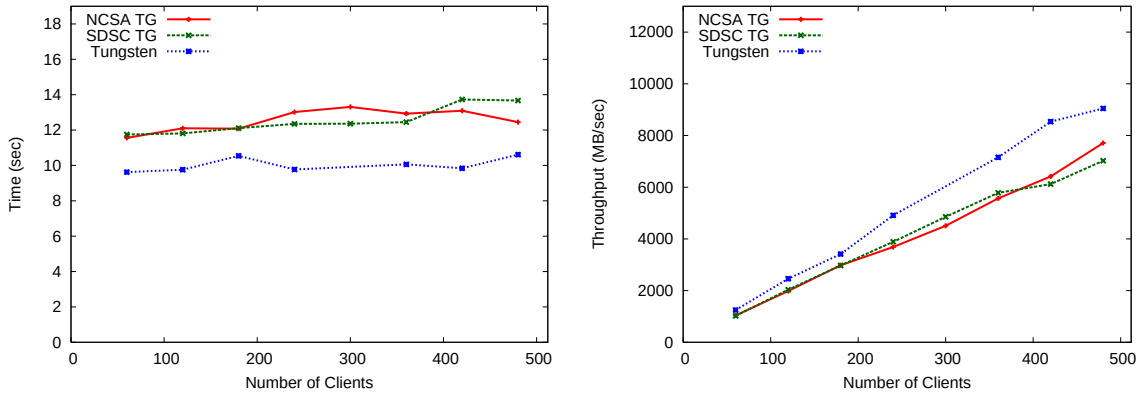


20:1 configuration

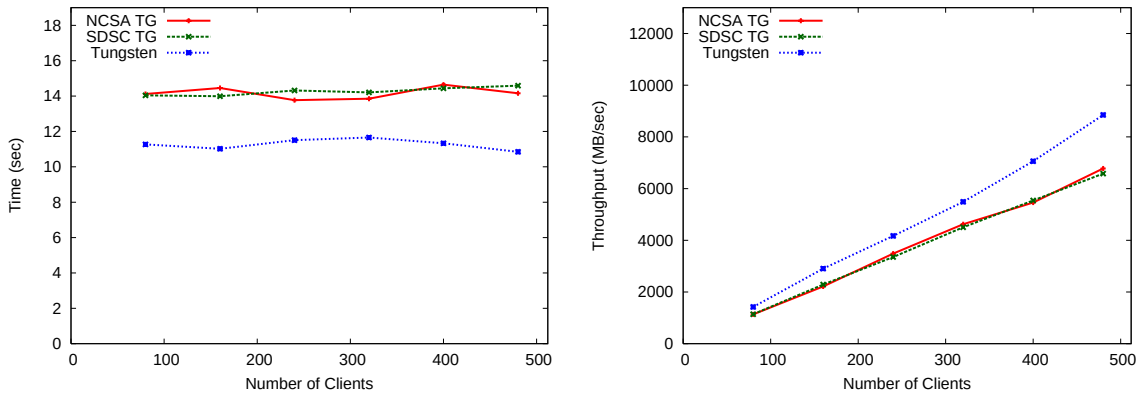


40:1 configuration

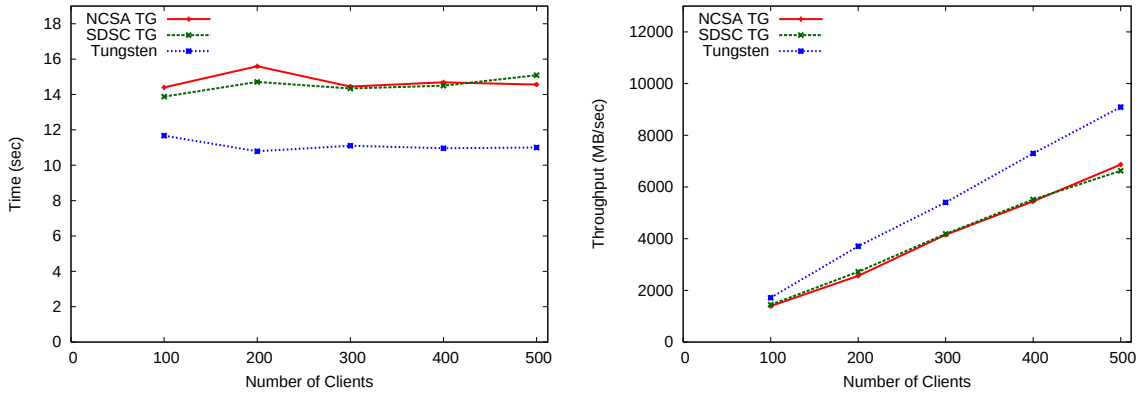
Figure 6.2: Checkpointing times and throughput.



60:1 configuration

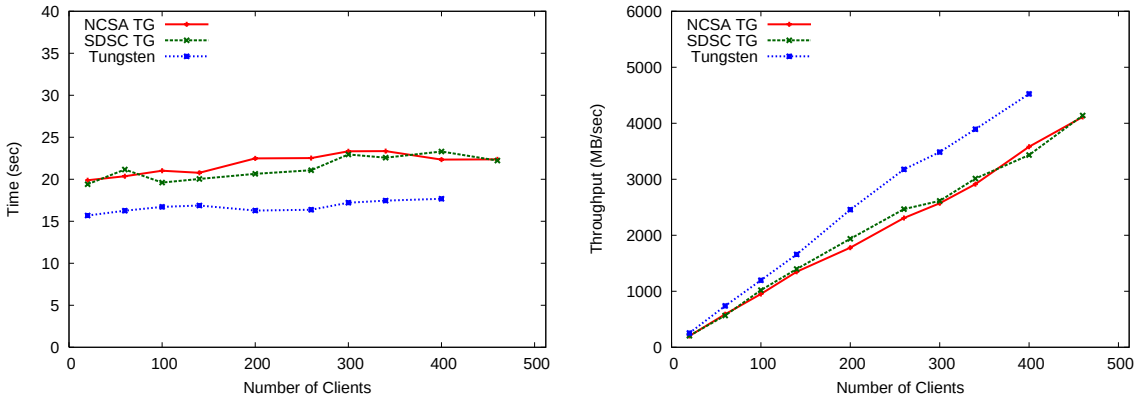


80:1 configuration

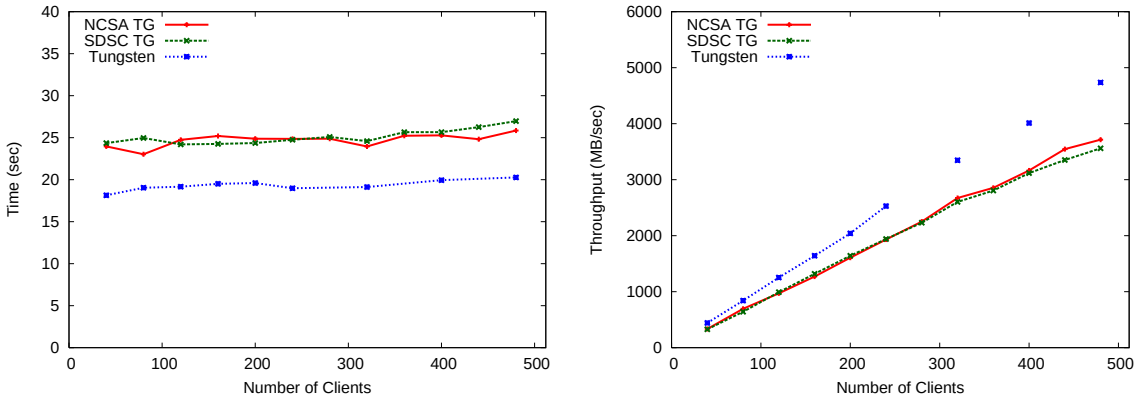


100:1 configuration

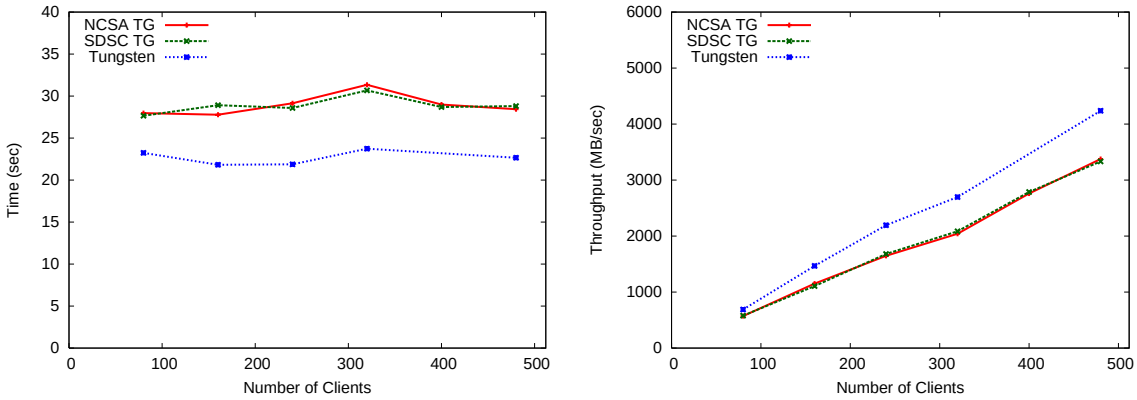
Figure 6.3: Checkpointing times and throughput.



20:2 configuration

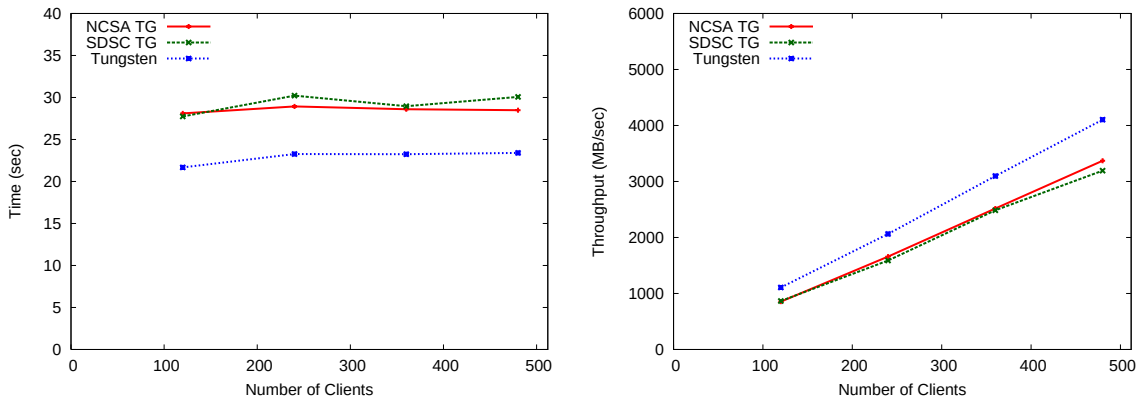


40:2 configuration

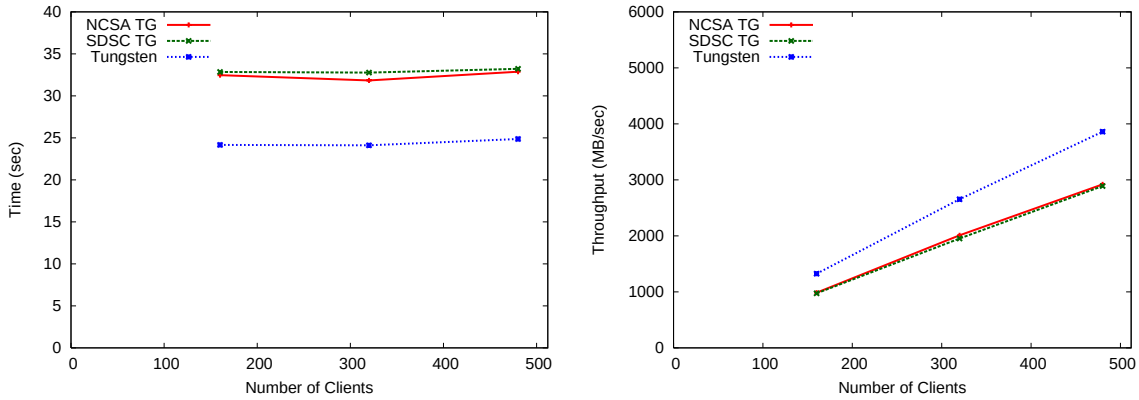


80:2 configuration

Figure 6.4: Checkpointing times and throughput.

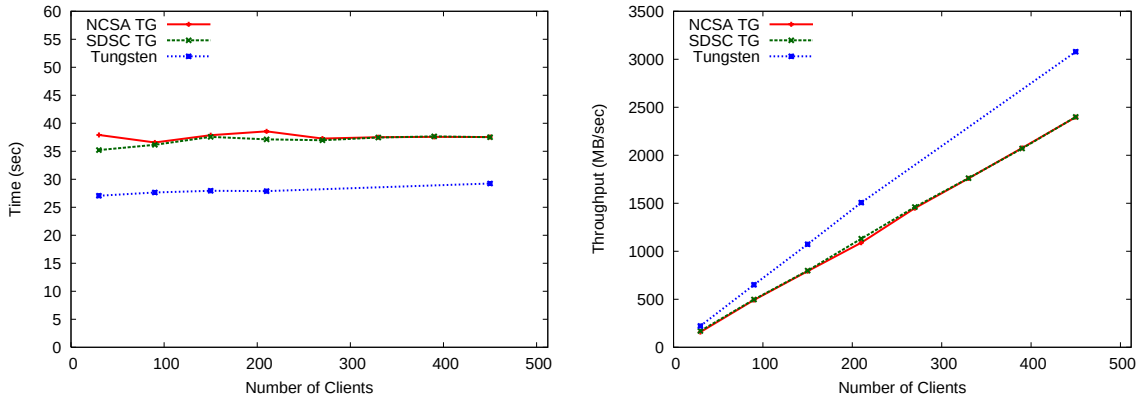


120:2 configuration

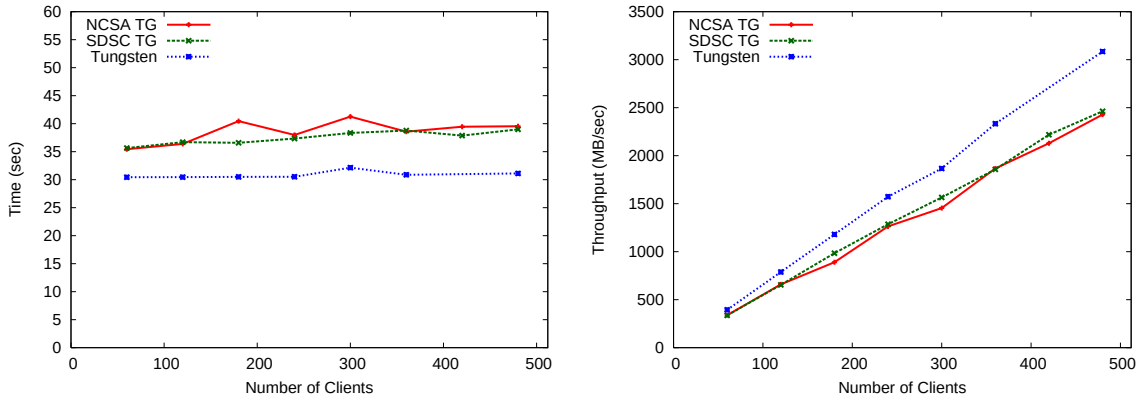


160:2 configuration

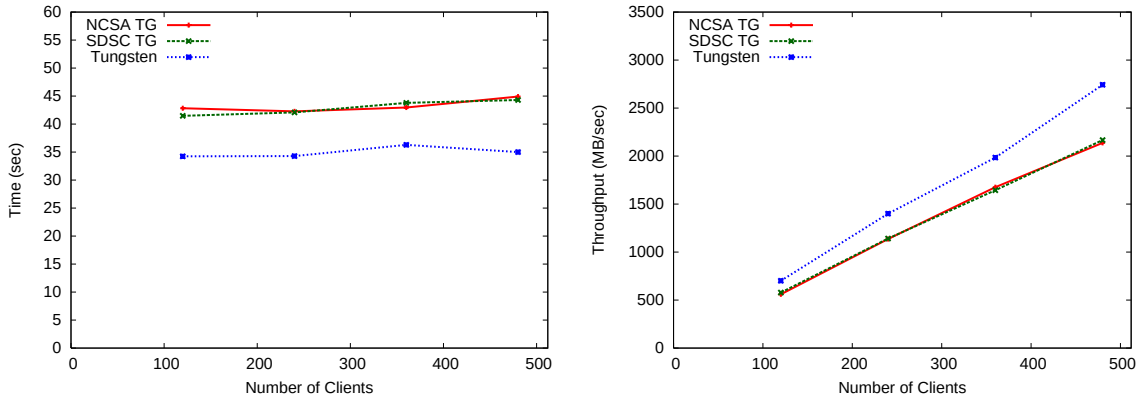
Figure 6.5: Checkpointing times and throughput.



30:3 configuration



60:3 configuration



120:3 configuration

Figure 6.6: Checkpointing times and throughput.

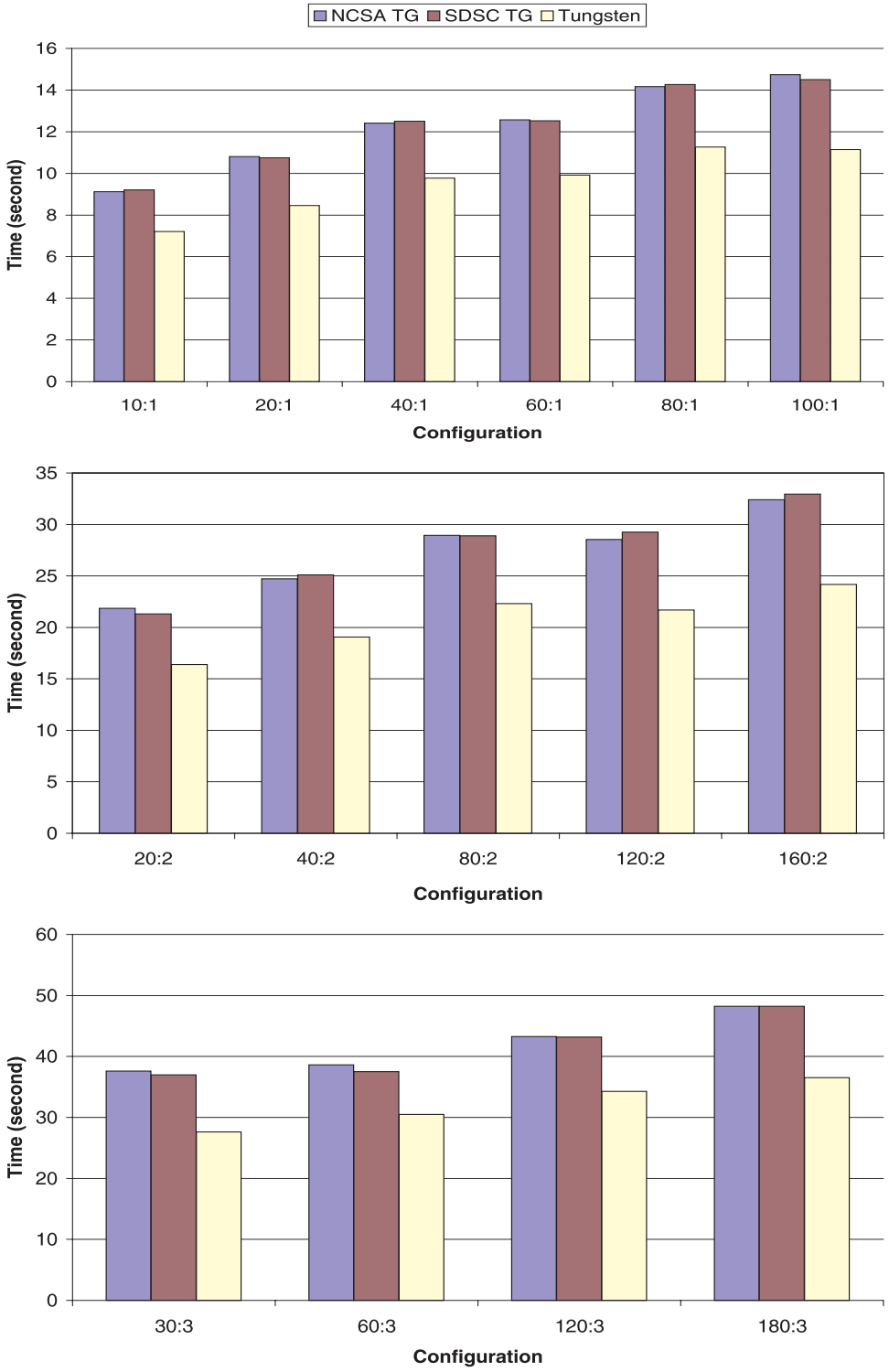


Figure 6.7: Checkpointing times comparison of one, two, and three spares per group.

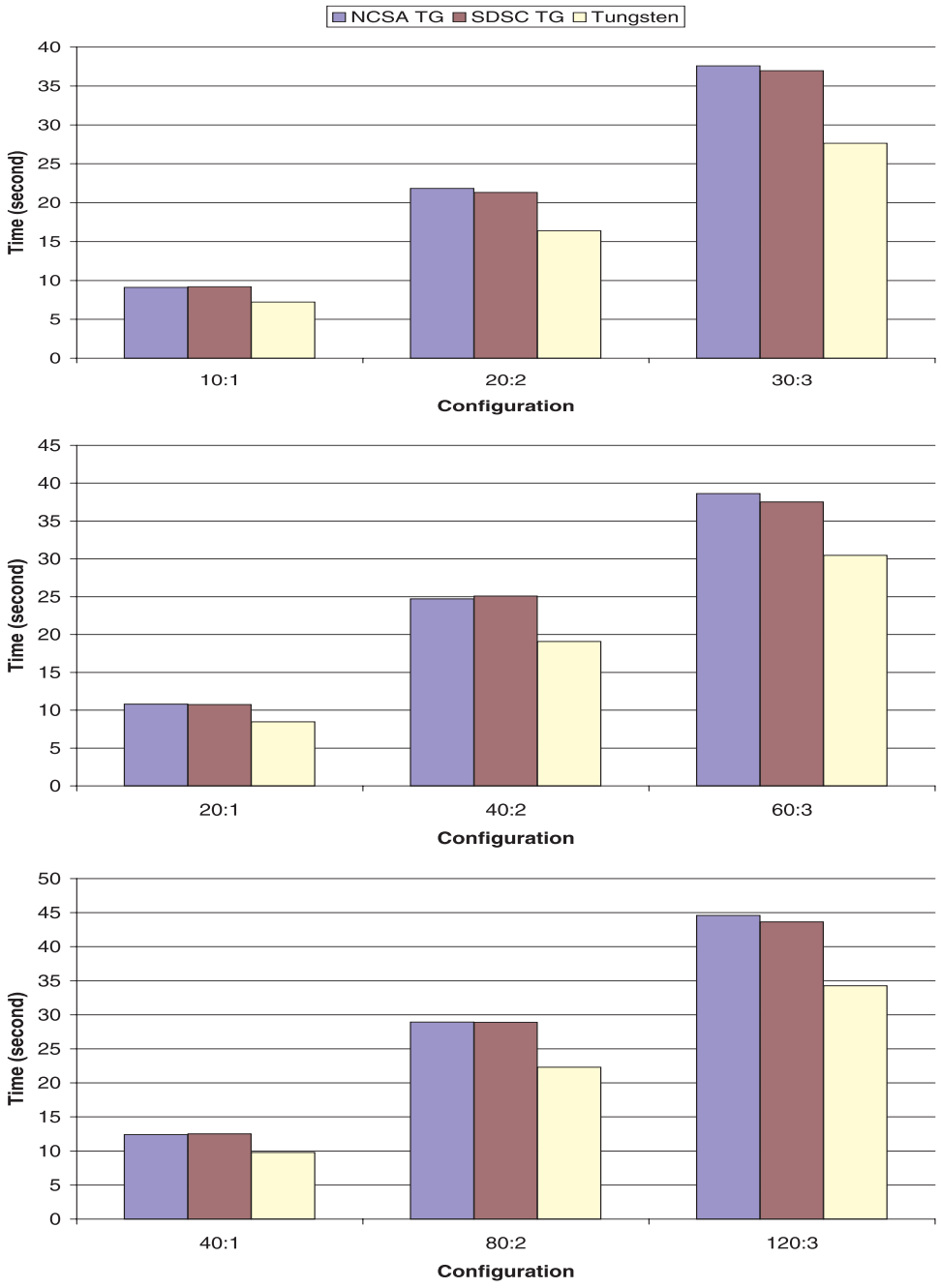


Figure 6.8: Checkpointing times comparison of merging two or three groups into one larger group.

6.3 Comparison with Disk-based Parallel File Systems

To show the better performance of diskless checkpointing, we also benchmarked the disk-based parallel file systems as a contrast.

6.3.1 The Parallel File Systems

The NCSA TeraGrid system provides three kinds of cluster-wide scratch storage for users. The first is the Network File System (NFS), which has 1 TB size and hosts user home directories. The second and the third both run IBM's General Parallel File System (GPFS) [43] on a Storage Area Network (SAN) composed of IBM TotalStorage FAStT 900 storage servers, but the connections to SAN are different. The `/gpfs_scratch1` directory is mounted in the Network Shared Disk (NSD) mode, in which 32 nodes are dedicated to act as the conduit to SAN, an array of eight FAStT servers with 39 TB of space. The `/gpfs_sanscratch` directory operates in the Directly Attached (DA) mode, in which every node that mounts this directory must connect directly to SAN via Fiber Channel. The SAN for DA mode consists of 24 FAStT servers and provides 90 TB of space. DA mode delivers performance several times better than NSD mode, but due to the direct connection requirement, currently only the phase-one nodes (1.3 GHz) can access it. For fail-over, all the FAStT servers are arranged in a sister pair configuration: each server node serves a set of primary disks, which has a backup node that can take over transparently for those disks and vice versa.

The NCSA Tungsten system has a dedicated sub-cluster of 104 I/O nodes connected to a SAN via Fiber Channel. The SAN is composed of fourteen DataDirect model 8000 storage servers, providing a total of 122 TB of space. The compute nodes are connected to I/O nodes via Gigabit Ethernet. The cluster-wide scratch storage uses the proprietary Lustre File System (CFS, mounted under `/cfs/scratch`) developed by Cluster File System, Inc [44]. CFS is a scalable parallel file system designed exclusively for large commodity clusters. Unlike traditional file systems which manage files on a block basis, CFS organizes all data related to a file as store objects. There is no hierarchical namespace but only a flat collection of store objects with distinct identifiers. Such

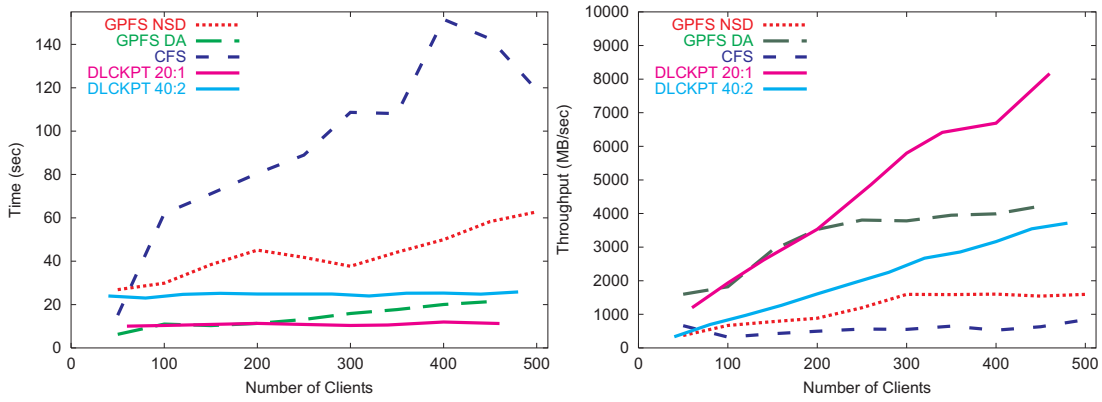


Figure 6.9: Disk-based file system performance comparison.

an object-based storage architecture can ease access control and storage management of data on clusters with high volume demand [45].

6.3.2 Results

We used the same micro-benchmark code mentioned in §6.2 and compared the performance of GPFS and CFS to DLCKPT on the NCSA TeraGrid. The result is shown in Figure 6.9. It can be readily seen that on the two high-performance computing systems we experimented, their disk-based parallel file systems cannot scale beyond a certain point, e.g., 300 clients in GPFS. GPFS in DA mode is 2.5-4 times better than in NSD mode and CFS is the worst of three disk file systems. Under 200 clients, GPFS DA performs equally well as DLCKPT 20:1 configuration does. We believe the reason that GPFS DA fared well in this case is the micro-benchmark code does not perform an `fsync()`, which will force the transfer of content of write cache to the disk, after each `write()`.

Nevertheless, DLCKPT starts to outperform all three disk file systems when the number of clients gets larger. Even the 40:2 configuration, which only has half of 20:1's throughput, catches up with GPFS DA at about 500 clients. We can conjecture that the root of unscalability of disk-based file systems lies in its inability to dynamically expand I/O nodes and I/O servers to cope the user's needs, whereas our DLCKPT has no such constraints.

6.4 Application Checkpointing Performance

In addition to micro-benchmarks, we also used two real data-intensive parallel codes to evaluate diskless checkpointing. Both codes are part of the ASCI Purple Benchmarks, representing the bulk of workload on the ASCI supercomputers.

6.4.1 The sPPM Application

sPPM [4] is a computational kernel used by many applications. It solves a 3-D gas dynamics problem on a uniform Cartesian mesh using a simplified version of Piecewise Parabolic Method. Its algorithm progresses in "double timesteps". Each double timestep involves 13 different directions of sweeps through the meshed data. Message-passing were largely used to update ghost cells from neighboring domains.

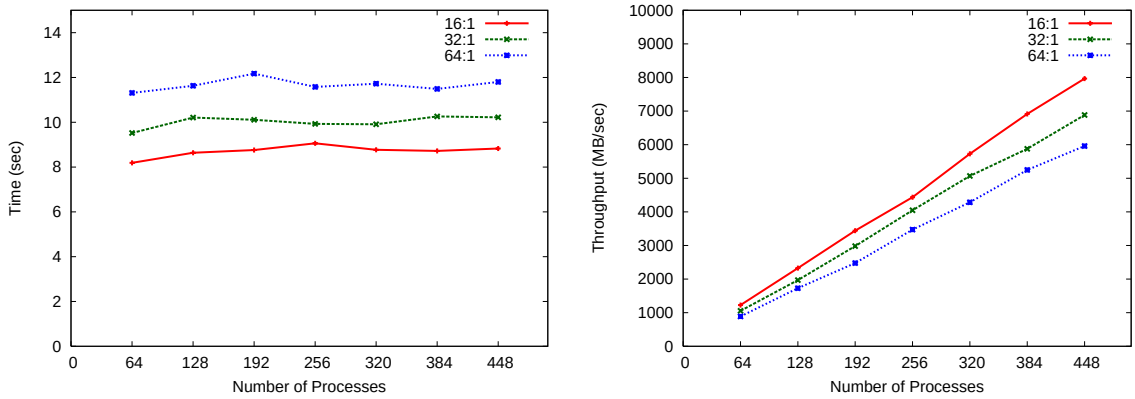
6.4.2 The Sweep3D Application

Sweep3D [5, 6] is the kernel of solver for 3-D time-independent particle transport problem using a multi-dimensional wavefront algorithm. Sweep3D exchange messages among processes as wavefront propagates diagonally across its 3-D space in eight directions.

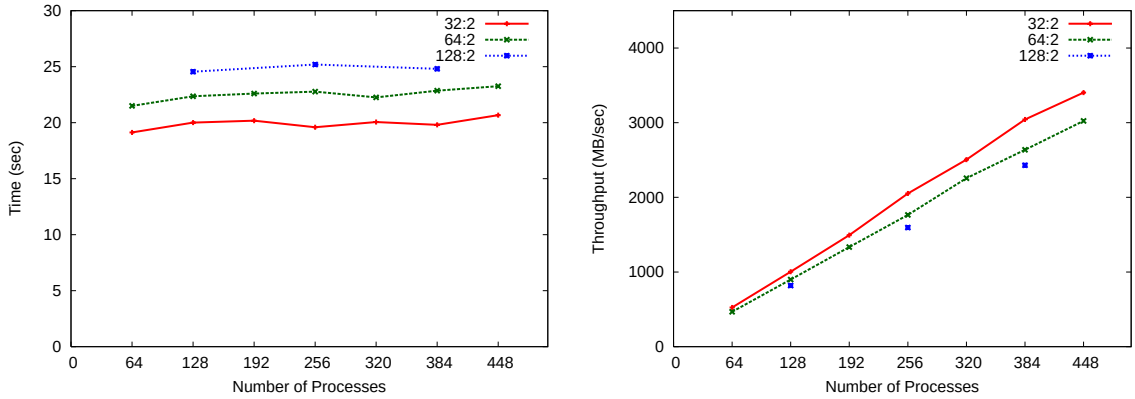
Both codes follow the Bulk Synchronous Parallel (BSP) computing model which characterizes process execution as iterations of supersteps. A superstep consists of, in order, local computation, global communication, and a barrier synchronization. Checkpoint can be selectively dumped at the end of a superstep. In both codes all processes dump checkpoint to their individual files and there is no sharing among the files during recovery.

6.4.3 Results

The NCSA TeraGrid system is the testbed of choice for this set of experiments. We configured both codes to checkpoint after every iteration. For sPPM (in single-precision mode) an iteration lasts for one minute and each process creates a 157 MB file, and for Sweep3D the time is half minute and



One spare per group



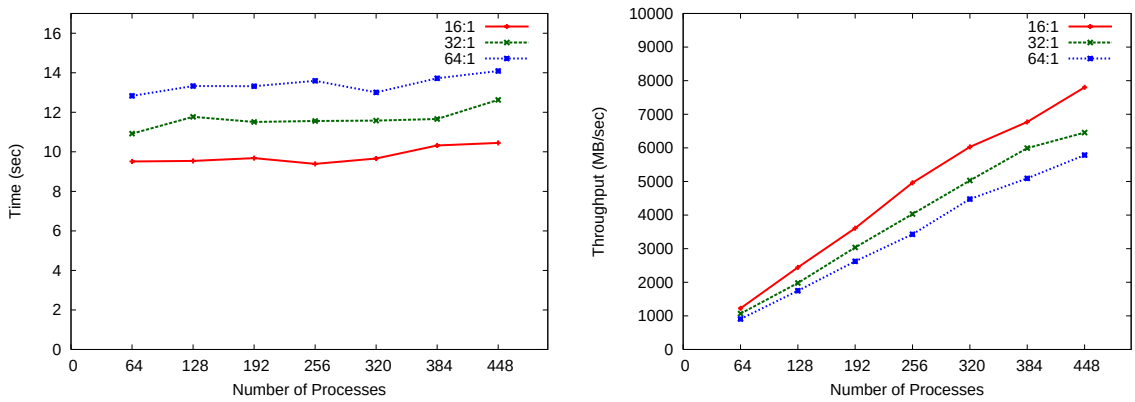
Two spares per group

Figure 6.10: sPPM checkpointing performance.

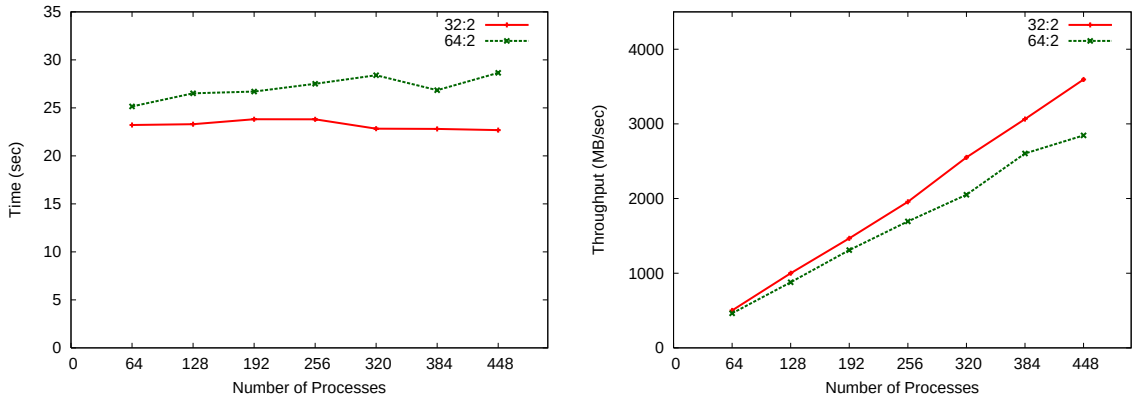
file size is 182 MB.

Figure 6.10 and 6.11 show the result. Both codes exhibit comparable performance to the micro-benchmark result in §6.2. In one spare per group case, both achieve 6-8 GB/s of throughput at 448 clients, and in two spares per group both have 3-3.5 GB/s of throughput.

We can infer that diskless checkpoint has a positive impact to the performance (i.e. time-to-solution) of both applications because it consumes almost constant time (8-25 seconds in sPPM and 9-27 seconds in Sweep3D) regardless of the number of processes, while the disk-based checkpoint could use as much as two minutes (based on results in §6.3) when there are 500 processes.



One spare per group



Two spares per group

Figure 6.11: Sweep3D checkpointing performance.

6.5 Application Recovery Performance

Application recovery has three components: process recovery, data recovery, and additional overhead. Process recovery is re-spawn the failed process on a spare, which is taken care by LA-MPI. Data recovery is performed by DLCKPT codec module. The additional overhead is induced by the launch of DLCKPT codec module and the communication between DLCKPT codec module and LA-MPI. As mentioned in §5.5, the DLCKPT codec module uses a high-performance MPI library that lacks self-recovery feature, so a node failure can cripple the DLCKPT codec module and a re-launch is required.

We tested the same applications with a 16:1 configuration. We wrote a simple fault injector to kill both a user process and the DLCKPT codec module to simulate a node failure. The injector is activated after the first checkpoint and before the second checkpoint. When it is activated, it sends SIGKILL signals to one of user processes and a DLCKPT codec module process.

The recovery results are presented in Figure 6.12 and 6.13. Data recovery performance is the same as the checkpointing performance, as expected. Process recovery is also quite scalable and takes no longer than 2.5 seconds. However, the additional overhead is the dominant part and accounts for up to 70 percent of total recovery time. Besides, its time increases along with the total number of processes, making recovery unscalable. The main cause lies in the extremely inefficient way the MPICH launches an MPI application.

Following the node list, the MPICH uses Secure Shell (ssh) utility to login to each node *in order* and create processes there. The linearity is already inefficient, let alone that ssh itself is also slow because it goes through a complex user authentication protocol. It is not an exaggeration that job launching times can take tens of minutes on very large parallel systems [41, 46].

The job launching time can certainly be improved. The makers of the MPICH developed an alternative, high-performance job launching facility called Multiple Purpose Daemon (MPD) [47]. There are also other research projects aimed at building fast and reliable job management systems with improved algorithms (e.g. binary tree) and interconnect support (e.g. multicast). STORM [46]

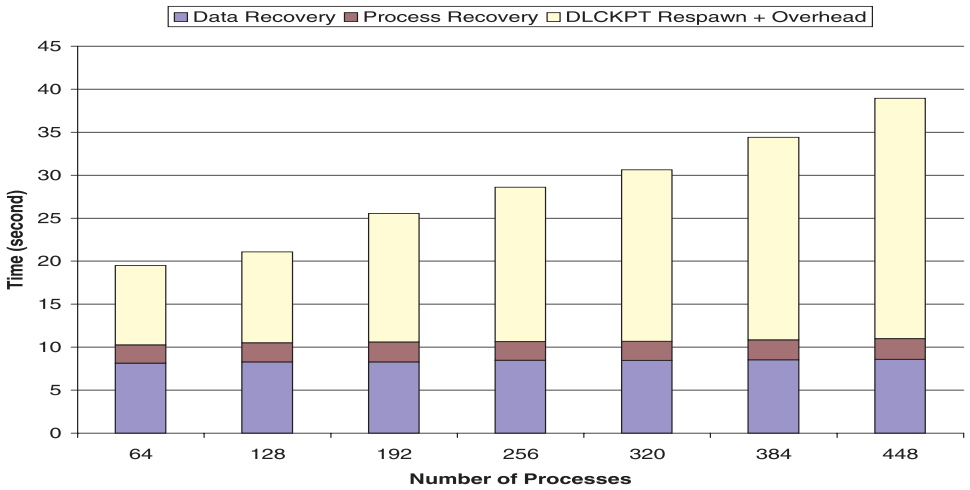


Figure 6.12: sPPM recovery performance.

exploits hardware barrier and broadcast available on Quadrics interconnect to achieve 0.11 seconds of launching a 12 MB job on 64 nodes. Projects like SLURM [48] and BProc [49] also reported comparable results.

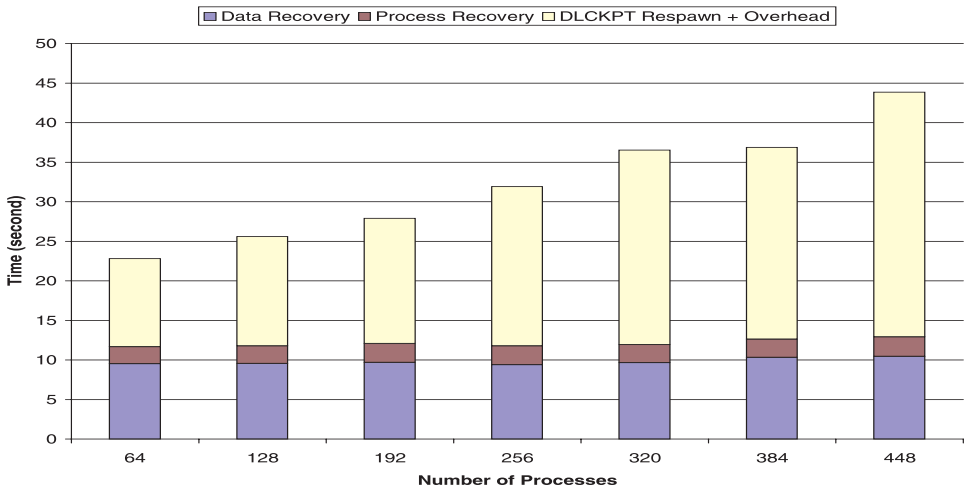


Figure 6.13: Sweep3D recovery performance.

6.6 Checkpoint Compression

Volatility is not the only shortcoming of using RAM as storage: another constraint is the small size compared to disks. In practice, when a user program checkpoints, it cannot simply overwrite the previous checkpoint because a failure can occur during checkpoint. Instead, the user program must write to a different file and delete the previous checkpoint until the current checkpoint is complete. Therefore, the total memory required could triple the process's memory.

Memory compression has become a feasible solution to increase the "effective" capacity as speed of modern processors is fast enough to squeeze the content data in real-time. Compression can also maximize the effective bandwidth because less data is transferred. As an example, IBM developed a novel memory controller architecture called Memory Expansion Technology (MXT) [50] to perform real-time compression and decompression of data traffic between the cache and the main memory. Diverse cache compression schemes are also extensively studied and evaluated [51, 52].

To assess the benefit of memory compression, we adopted a lossless compression library called LZO [53]. Its compression algorithm is a derivative of the industrial-strength Lempel-Ziv compressor. LZO is designed to favor speed over compression efficiency to achieve real-time compression and decompression.

We added LZO to the application code instead of the DLCKPT file system in order to keep our DLCKPT code clean and simple. Because LZO is a block compressor, we modified the way the application allocates the memory such that the portion of the application address space to be dumped during a checkpoint is a continuous block. Then in a checkpoint, the data is first compressed and then written to the DLCKPT file system.

There are two gauges of compression efficiency: "compression ratio," defined as compressed size divided by the original size, and "compression time". The definition of compression ratio may seem awkward because a higher compression ratio corresponds worse compressibility, but it can correlate nicely to compression time, as we should see later from the experimental results.

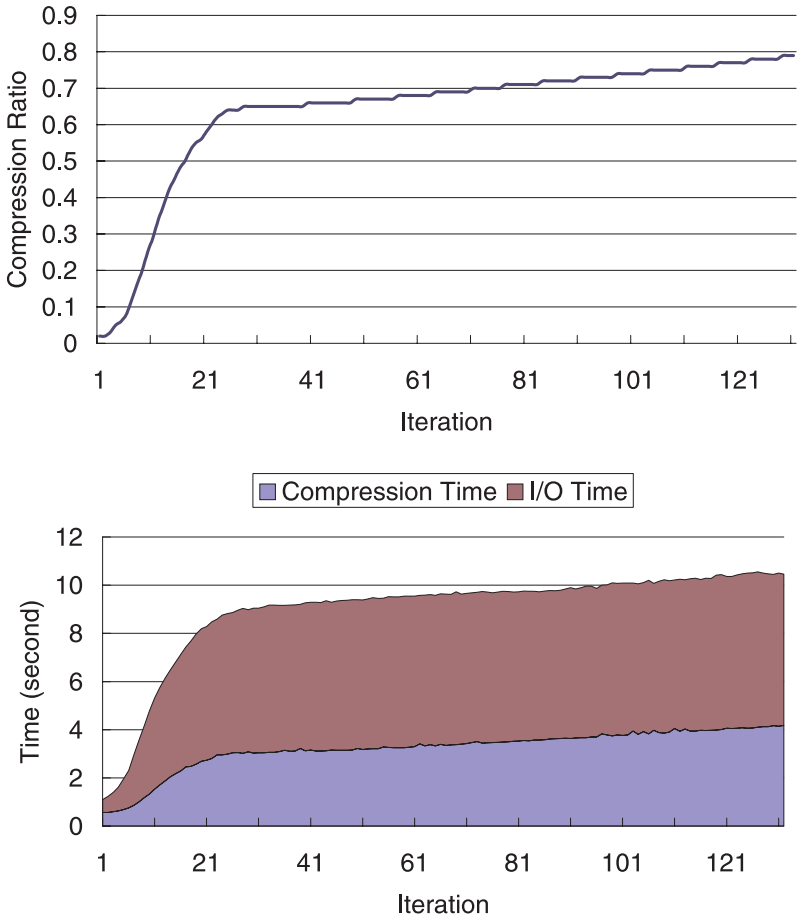


Figure 6.14: Compression ratio and combined compression and I/O time of sPPM.

We tested with sPPM and Sweep3D using 16 MPI processes and 16:1 configuration on the NCSA TeraGrid system. Our result shows that Sweep3D is very incompressible; the compression ratio is greater than 0.8 across all processes. This corroborates the findings of Alameldeen and Wood [51]. They tried several SPEC2000 and commercial benchmarks and found that floating-point benchmarks are generally less compressible (with compression ratios 0.77 to 1) than their integral counterparts (ratios 0.4 to 0.7).

The result of sPPM is more interesting. Figure 6.14 presents the plots of compression ratio and time as a function of iteration step. The result is chosen from one of the processes, but all processes exhibit the same behavior: both compression ratio and time increases as the application

progresses. In the first 10 iterations, the compression time is 0.6-1.2 seconds. After 30 iterations, the compression time grows to a point where the combined (compression and I/O) time is longer than 9 seconds, the time if no compression is used.

The growing incompressibility is also observed in Sweep3D. This phenomenon can be explained from the viewpoint of information entropy. The principle of compression is to replace repetitive data patterns with compact representation. Initially, the data is highly homogeneous (e.g. all zeros) and thus highly compressible. After iterations of computations and changes on data, the information content in data becomes less regular and hence less compressible.

The concomitant growth of compression time is related to the dictionary-based compression algorithm of LZO. LZO maintains a sliding dictionary of frequent strings and their encodings. If a stream of bytes matches an entry in the dictionary, the output is the corresponding encoding; otherwise, the stream is output in it original form with an appropriate prefix. For highly compressible data, the repetitive strings can be quickly identified in dictionary and hence the short compression time. If the data is pattern-less, then full search and update of dictionary will be commonplace and waste more CPU cycles.

6.7 Summary

We used a micro-benchmark and two real parallel scientific applications on large high-performance computing systems to assess the efficacy of diskless checkpointing. The results showed great scalability of diskless checkpointing, as expected. When the number of clients grows to a certain point, the diskless checkpointing starts to outperform the disk-based parallel file system. In one spare per group configuration, the throughput of 448 clients can reach 9-12 GB/s, and in two or three spares per group configuration, the throughput of the same number of clients can achieve 3.6-6 GB/s and 2-3.8 GB/s, respectively.

We also measured the recovery performance and found the dominant factor is the job re-launching time of the DLCKPT codec module. If this factor is ruled out, the recovery is also highly scalable and its performance is comparable to the checkpointing.

Finally, because the memory storage is scarce compared to disks, we experimented with checkpoint compression using a real-time lossless compression library. The result shows that compression can effectively reduce the application checkpoint size and hence the I/O time. However, after more iterations of computation, the application checkpoint becomes less compressible, so the incurred compression time overhead outweighs the advantage it brings.

Chapter 7

Performance Analysis

The goal of this chapter is to develop a performance model of of the DLCKPT system and derive its model parameters from the experimental results presented in the previous chapter. In reality there are so many affecting factors (e.g. communication middleware optimization, node affinity [see §6.1.2], OS interference [54], and interconnection architecture) that it is unlikely to correctly predict the performance of a system one or two orders greater than the PC clusters on which we experimented. Despite the difficulty, we propose a tractable model that can explain most experimental results and is reasonable enough to predict performance on larger systems.

7.1 Overview

Our model consists of several components, each of which corresponds to a stage of execution. For checkpointing, the model is broken up into the following components/stages:

1. The user application writes to the DLCKPT file system. Multiple checkpoint files can be written at this stage.

2. All user processes finish writing and enter a barrier. After the barrier, the root process of the user application prompts the DLCKPT codec module to start encoding.

3. The DLCKPT codec module runs Reed-Solomon encoding if necessary.

4. The DLCKPT codec module uses `MPI_Reduce()` to produce redundancy codes.

5. The DLCKPT codec module finishes the reduction and enter a barrier. After the barrier, the root process of the DLCKPT codec module informs the user application of completion.

Therefore, the checkpoint time for one spare per group is

$$C_1 = MemCopyTime + ReductionTime + MultiGroupOverhead$$

and for $k > 1$ spares per group is

$$C_k = MemCopyTime + k \cdot (RSEncodingTime + ReductionTime) + MultiGroupOverhead$$

The $MemCopyTime$ term corresponds to stage 1 (see §7.2), $RSEncodingTime$ to stage 3 (see § 7.3) and $ReductionTime$ to stage 4 (see § 7.4).

The stage 5 is critical to the scalability of diskless checkpointing on large systems. The reported performance in the previous chapter is based on the time the *last* group entering the barrier (i.e. last one that completes in stages 3 and 4.) However, based on our measurement, not all groups finishes stages 3 and 4 in exactly the same time, even if they all have the same amount of data to process. Therefore, we use the $MultiGroupOverhead$ (see §7.5) term to account for the additional overhead due to multiple groups.

Recovery follows essentially the same scenario except that the LA-MPI must spawn the DLCKPT codec module ($JobLaunchingTime$), must restart the failed process on a spare and all processes must reload the checkpoint (combinedly $ProcessRecoverTime$.) So we derive the recovery time for one spare per group scheme as

$$R_1 \approx FailureDetectionTime + ProcessRecoverTime + JobLaunchingTime + C_1$$

The recovery time for k ($k > 1$) spares per group is

$$R_k \approx FailureDetectionTime + ProcessRecoverTime + JobLaunchingTime + 2 \cdot C_k$$

R_k differs from R_1 in that when there are more than one spare per group, immediately after a recovery, the redundancy codes must be recomputed because old codes are no longer valid, so R_k has a $2 \cdot C_k$ term.

7.2 Memory Copy Performance

We wrote a benchmark to measure the efficiency of `memcpy()` on large data arrays. rate. On a machine with 3.2 GHz Pentium 4 Xeon processors, we got about 1000 MB/s with the GCC compiler and 1500 MB/s with the Intel C Compiler (ICC) version 8. On a machine with 1.3 GHz Itanium 2 processors, we got about 770 MB/s with GCC and 2000 MB/s with ICC. We believe the performance difference comes from a better implementation of `memcpy()` of ICC. At any rate, such speed is fast enough to deal with large checkpoints.

7.3 Reed-Solomon Encoding Performance

In §4.3.3 we mentioned our implementation of Reed-Solomon codes. Both encoding and decoding are simply a series of table look-ups that map each user data item in `source` into the result array `target`:

```
for (i=0; i < SIZE; i++)
  target[i] = RSTable[source[i]];
```

All of `source`, `target`, and `RSTable` are arrays of 16-bit unsigned short integers. The `RSTable` has 2^{16} elements and its size is 128 KB.

Although the above code seems simple, its performance can vary greatly. We have identified two bottlenecks that can affect the performance by a factor up to three: data locality and compiler optimization.

It has been well studied that different data patterns can yield different cache reuse rates and hence different access speed. In general, sequential accesses are more favorable than random accesses. To verify how data locality affects the performance of Reed-Solomon encoding, we performed a simple benchmark. We filled the the `source` array with different data patterns as follows:

Pattern	Fill Method
Constant	`source[i] = 0`
Less Random	`source[i] = rand() % 1024`
Random	`source[i] = rand() % 8192`
Fully Random	`source[i] = rand()`
Sequential	`source[i] = i`
Stride	`source[i] = 1+32*i`
Stride Random	`source[i] = 1+32*rand()`

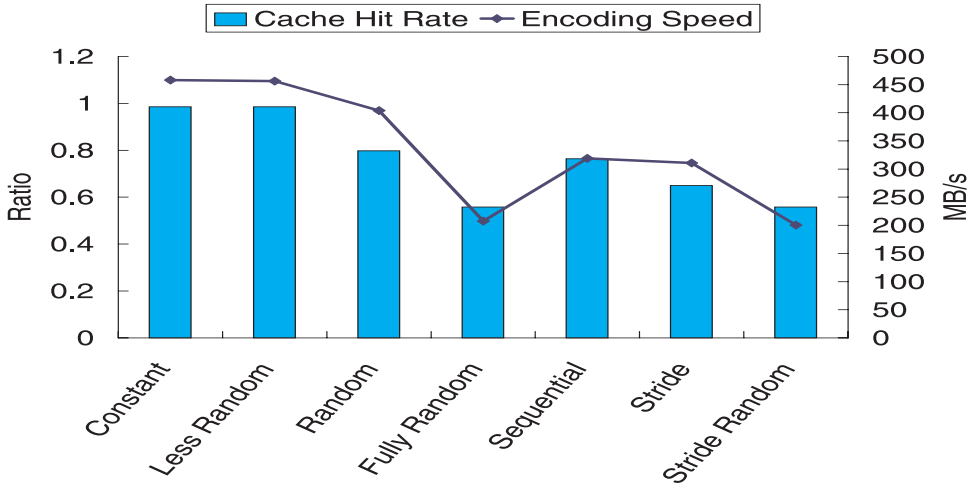
We took the processor cache into consideration when we designed the above patterns. The Itanium 2 has 16 KB level-one (L1) data cache and 256 KB L2 cache. Pentium 4 Xeon has 8 KB L1 data cache and 512 KB L2 cache. The cache line size in both processor' L1 cache is 64 bytes.

It is clear that `RSTable` cannot fit into both processors' L1 caches, and this implies that there may be quite a few cache misses for certain kinds of data patterns. The Stride pattern is to artificially reduce L1 cache reuse because `RSTable` is accessed every 32 elements, which are a single cache line's load.

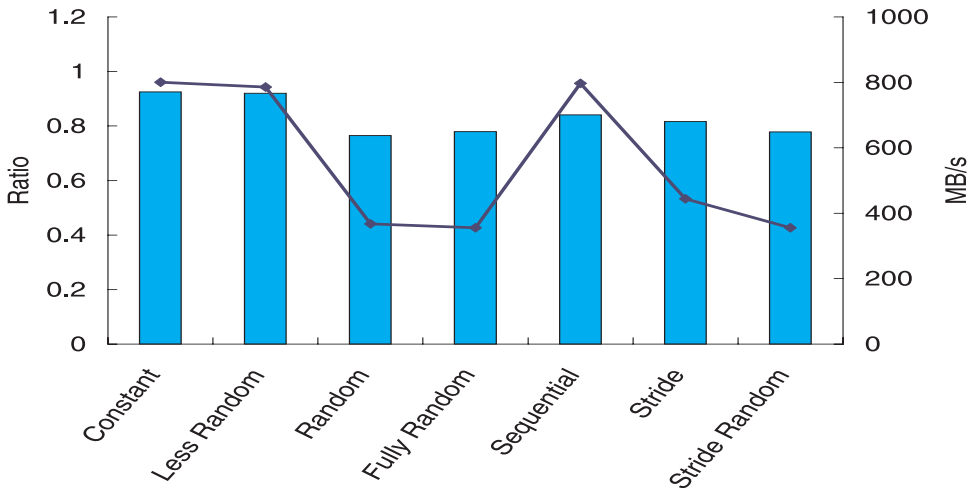
We used ICC with `-O3` flag to compile the benchmark. We also use PAPI [55] to get information on cache efficiency. Figure 7.1 shows the result. The encoding speed reaches its peak when the portion of `RSTable` in use can fit into L1 cache, e.g. Constant and Less Random. On the other hand, the speed drops to less than half (43%) of peak performance for two cache-unfriendly patterns: Fully Random and Stride Random. Overall, we found that the encoding speed is strongly correlated to the L1 cache hit rate in Itanium's case and less so in Pentium's case.

The second factor that affects performance is compiler optimization. We tested the GCC and ICC with varied optimization flags on Fully Random data pattern. The `-O0` flag indicates no optimization and `-O3` indicates the most aggressive optimization. Figure 7.2 presents the result. Here we had contradictory outcomes. The general belief is that ICC generates better code than GCC, and this can be validated in Itanium case: ICC outperforms GCC by 73% with both `-O3` flag set. Using PAPI, we found that GCC generates code that has three times the stalled cycles than ICC does. Further investigation shows that the GCC-generated code executed three times more branches.

The superiority of ICC reversed on Pentium 4. Here over-optimization causes inferior performance, even for GCC. Although we measured various performance metrics with PAPI, we could



(a) 1.3 GHz Itanium 2

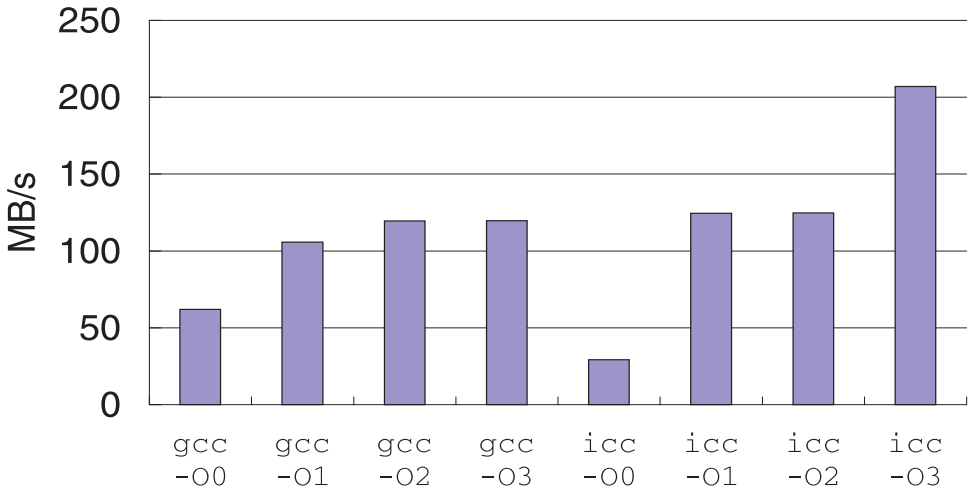


(b) 3.2 GHz Pentium 4 Xeon

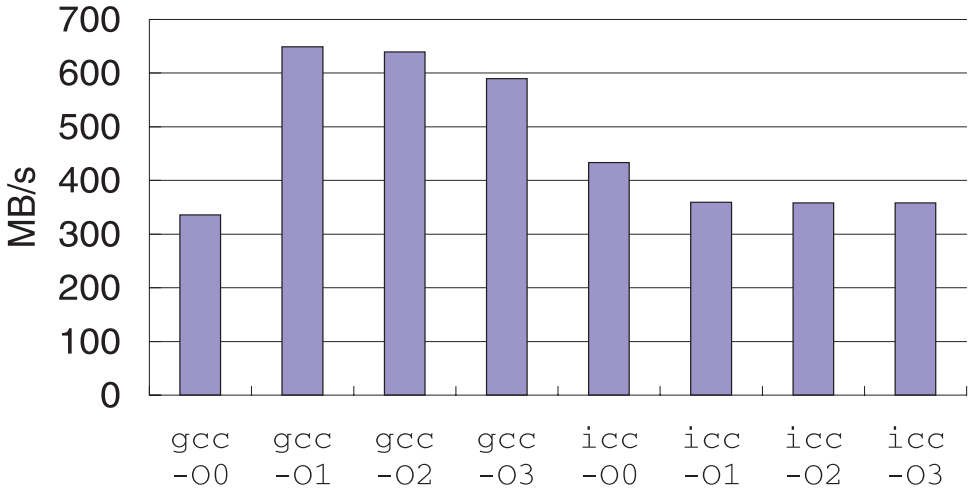
Figure 7.1: Reed-Solomon encoding performance and L1 cache hit rate for different data patterns.

not find a single metric that best explains this phenomenon. We suspect the architecture of Itanium relies more on the compiler to do optimizations, while the Pentium 4's already sophisticated built-in optimization can sometimes offset compiler efforts.

In practice no matter how the encoding code is optimized, on Pentium 4 the speed is fast enough to encode a large checkpoint in a second or two. On Itanium the encoding process can be a bottleneck, especially when the code is not properly optimized. For example, the encoding speed is only 30 MB/s with ICC `-O0`, which is even slower than the bandwidth of high-speed networks like Gigabit Ethernet.



(a) 1.3 GHz Itanium 2



(b) 3.2 GHz Pentium 4 Xeon

Figure 7.2: Reed-Solomon encoding performance of different compiler optimizations.

7.4 Reduction Performance

As mentioned in §4.7, the efficiency of `MPI_Reduce` calls is the most critical to the overall performance of diskless checkpointing. In this section we examine two different `MPI_Reduce` implementations and their performance.

We used version 1.2.5 of MPICH library [24] and version 1.1.1 of ChaMPIon/Pro in our experiments. The 1.2.5 and earlier versions of MPICH employs a Binary Tree algorithm in the reduction operation [42]. Although we do not have access to ChaMPIon/Pro's source code, we suspect it is also based on the same algorithm.

Let n be the number of processes, $0, \ldots, n-1$ be the process ranks, and d be the checkpoint size. In the Binary Tree algorithm, at step i, process $(2^i * k)$ receives d amount of data from process $(2^i * k + 2^{i-1})$ and performs reduction (k runs from 0 through $\lceil \lg n \rceil - i$). Figure 7.3 illustrates an example of four processes with summation as the reduction operation and P0 being the root process. Since there are $\lceil \lg n \rceil$ steps, the time complexity is $O(\lceil \lg n \rceil d)$.

At first glance, the Binary Tree algorithm seems to be the most efficient way of reduction. However, at time of writing we learned that the 1.2.6 version of MPICH library adopts an even faster algorithm called Rabenseifner's algorithm [42]. For long messages, this algorithm achieves $O(\frac{(n-1)d}{n})$ and can be further bounded above by $O(d)$ for large n. The idea behind Rabenseifner's algorithm is very simple: the reduction is implemented as a reduce-scatter followed by a gather operation. Both reduce-scatter and gather operations are defined in the MPI specification as `MPI_Reduce_scatter` and `MPI_Gather` functions. Reduce-scatter is a variant of reduce, in which the send buffer at each process is a vector of n blocks and the result of reduction is scattered over all processes. Gather is to collect data from all processes to the root process. Figure 7.4 demonstrates Rabenseifner's algorithm at work.

The key to the high performance of Rabenseifner's algorithm lies in the implementation of reduce-scatter and gather. MPICH 1.2.6 adopts the Pairwise Exchange algorithm for reduce-scatter operation over messages longer than 512 KB [42]. In the Pairwise Exchange algorithm, at step i,

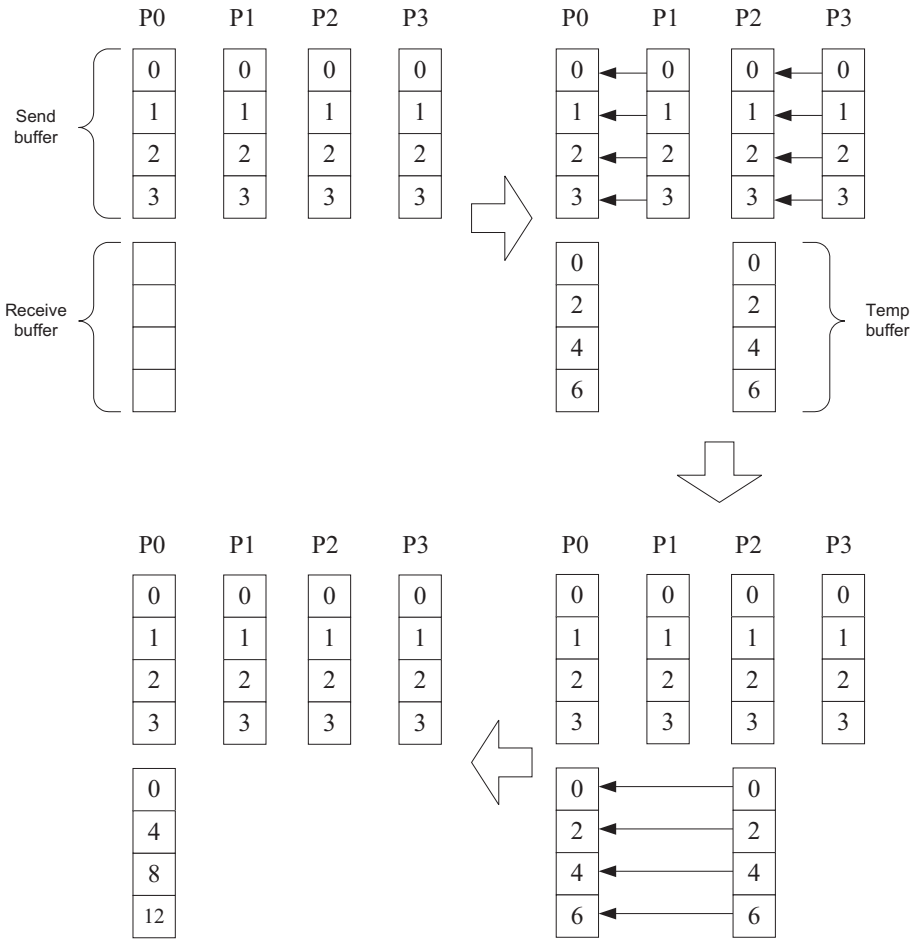


Figure 7.3: Binary Tree algorithm for reduction.

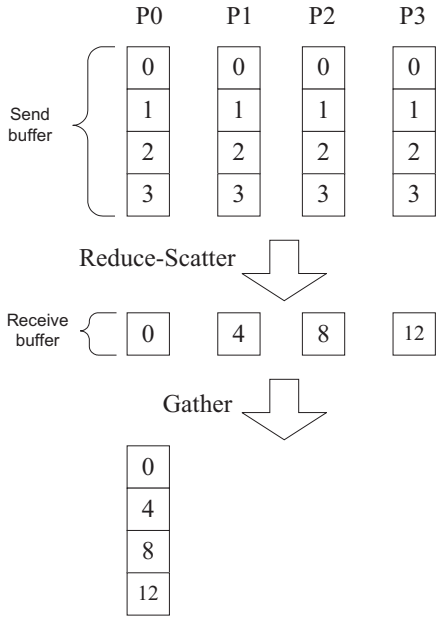


Figure 7.4: Rabenseifner's algorithm for reduction.

process k sends data to process $k + i$ and receives data from process $k - i$ (with wrap-around) where k runs from 0 through $n - 1$. The data sent is only the data needed for the scattered result on the receiving process. Thus, only $\frac{d}{n}$ amount of data per process is sent in a step. The algorithm concludes after $n - 1$ steps. Assume messages are long and the latency (i.e. message set-up time) can be ignored, then the time complexity is $O(\frac{(n-1)d}{n})$. Figure 7.5 gives an example of the Pairwise Exchange algorithm.

MPICH 1.2.6 implements gather operation using a Binary Tree algorithm similar to the reduction algorithm in MPICH 1.2.5. Figure 7.6 shows the gather algorithm. Despite the seeming similarity, the time complexities are disparate. The time of gather does not grow *logarithmically* with respect to the number of processes n; instead, it is $O(\frac{(n-1)d}{n})$. A comparison of Figure 7.3 with 7.6 reveals the difference: for reduction, constant (in this case, four) units of transfer are required for every step, while for gather, the units of transfer start with one and grow exponentially.

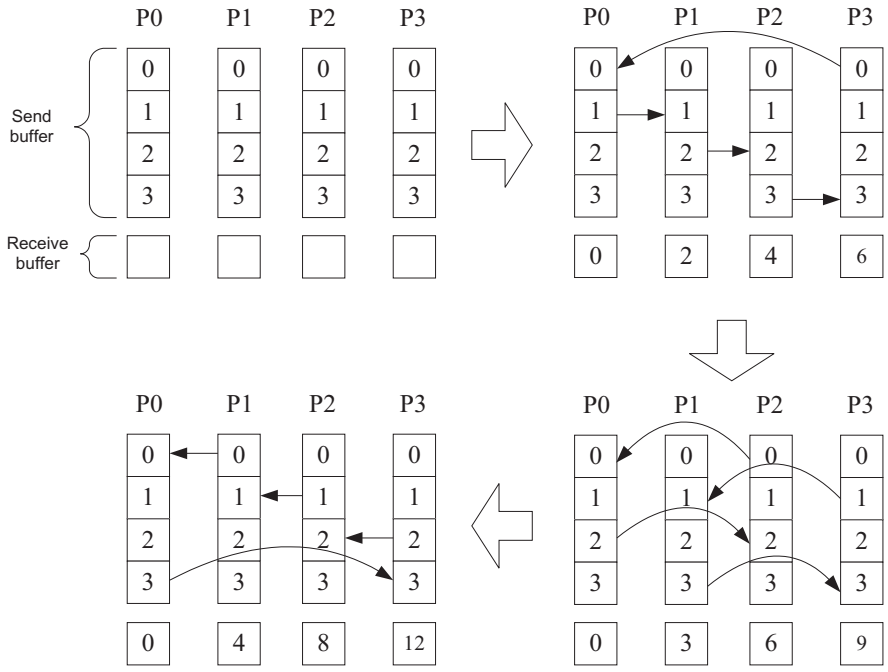


Figure 7.5: Pairwise Exchange algorithm for reduce-scatter.

Hence, the gather complexity is

$$\sum_{i=1}^{\lceil \lg n \rceil} \frac{d}{2^i} = \frac{(n-1)d}{n}$$

where $\lg n$ is base-2 logarithm function.

Combining reduce-scatter and gather operations, Rabenseifner's algorithm achieves $O(\frac{(n-1)d}{n})$ time complexity. Rabenseifner's algorithm is not only time-efficient but also space-efficient. In Binary Tree algorithm, half of the processes need to prepare a temporary buffer as large as the send buffer to hold intermediate results, while in Rabenseifner's algorithm the temporary buffer is no larger than half of the send buffer.

We measured the performance of old and new algorithms of `MPI_Reduce` on two systems (see § 6.1). We experimented 10:1, 20:1 and 40:1 configurations with one group and used bitwise XOR as reduction operation. The result is presented in Figure 7.7. Rabenseifner's algorithm is 1.5 times faster in most cases and twice as fast as Binary Tree algorithm for larger group and checkpoint size. A comparison of time growth of both algorithms is shown in Figure 7.8, which confirms that

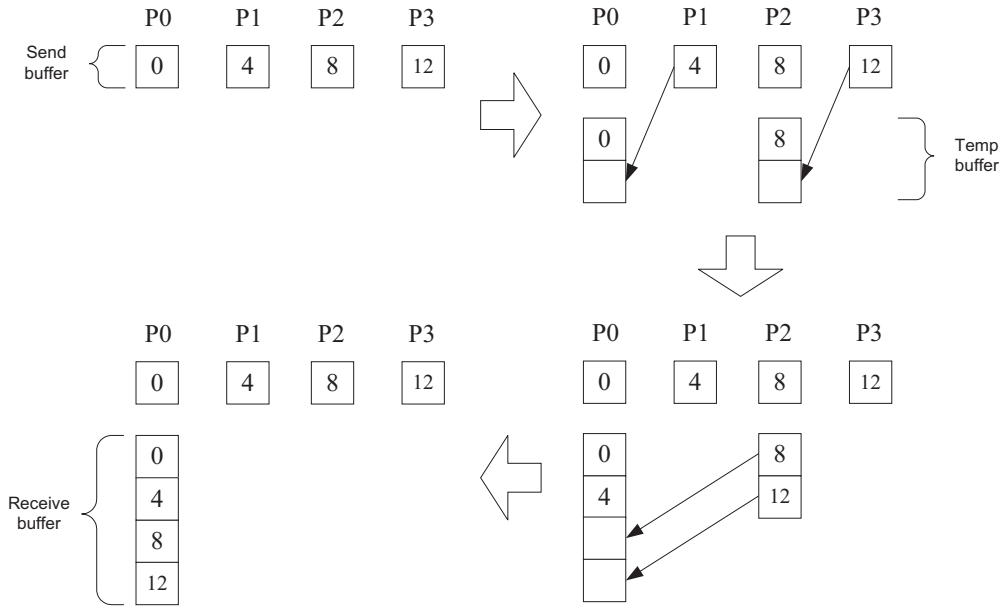


Figure 7.6: Binary Tree algorithm for gather.

Rabenseifner's algorithm has an almost constant execution time whereas the Binary Tree algorithm's time grows logarithmically with respect to group size.

Based on the above experimental data, we used the least-square curve-fitting technique to find parameters of performance models for old and new reduction algorithms. The checkpoint time of a single group with one spare on the Tungsten system with ChaMPIon/Pro library is:

$$(0.00785d - 0.07427)\lceil \lg n \rceil$$

on the TeraGrid system with MPICH-GM 1.2.5:

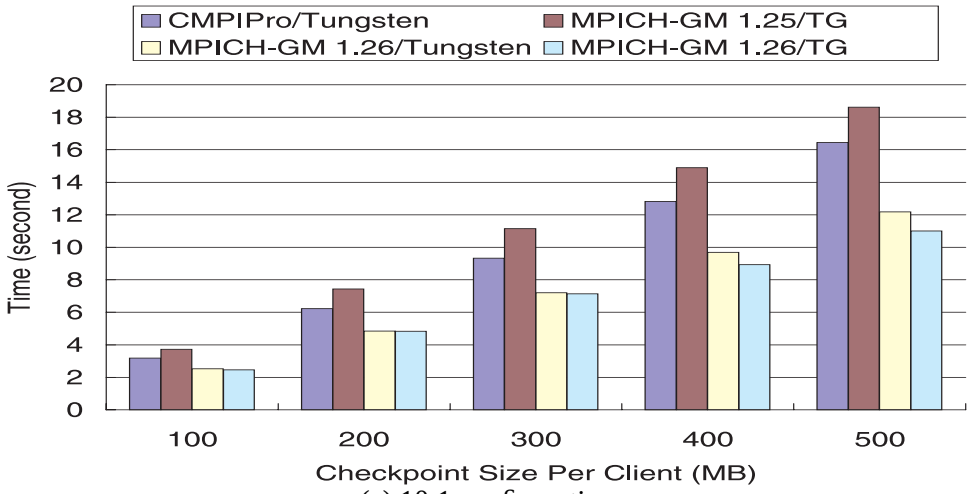
$$(0.00961d - 0.09926)\lceil \lg n \rceil$$

on the Tungsten system with MPICH-GM 1.2.6:

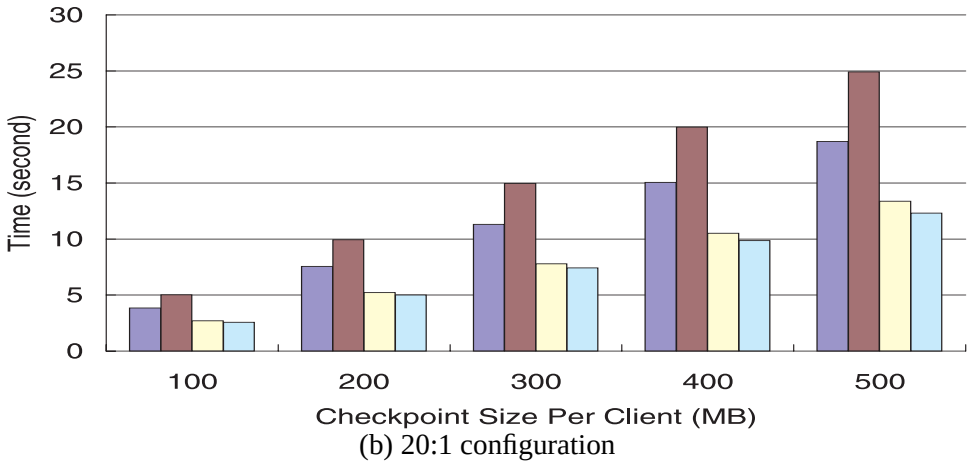
$$0.02781\frac{(n-1)d}{n} - 0.1281$$

and on the TeraGrid system with MPICH-GM 1.2.6:

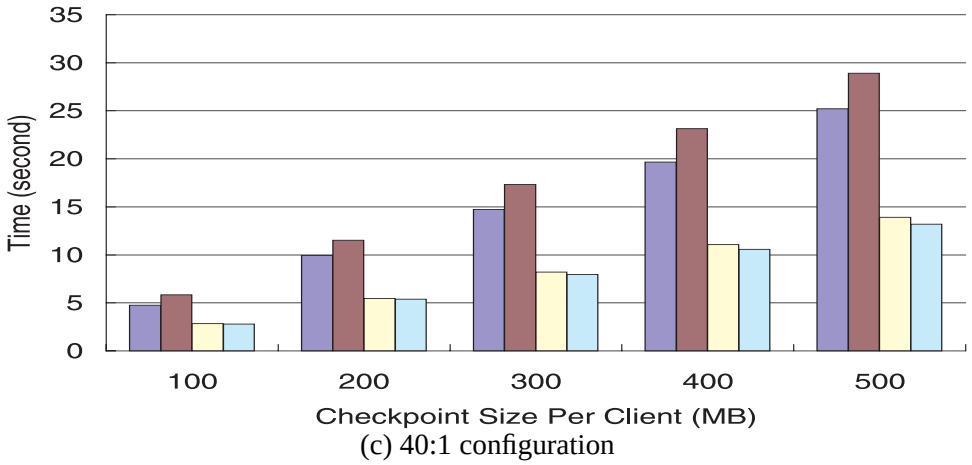
$$0.0254\frac{(n-1)d}{n} + 0.1734$$



(a) 10:1 configuration



(b) 20:1 configuration



(c) 40:1 configuration

Figure 7.7: Comparison of different reduction algorithms for different per-process checkpoint sizes.

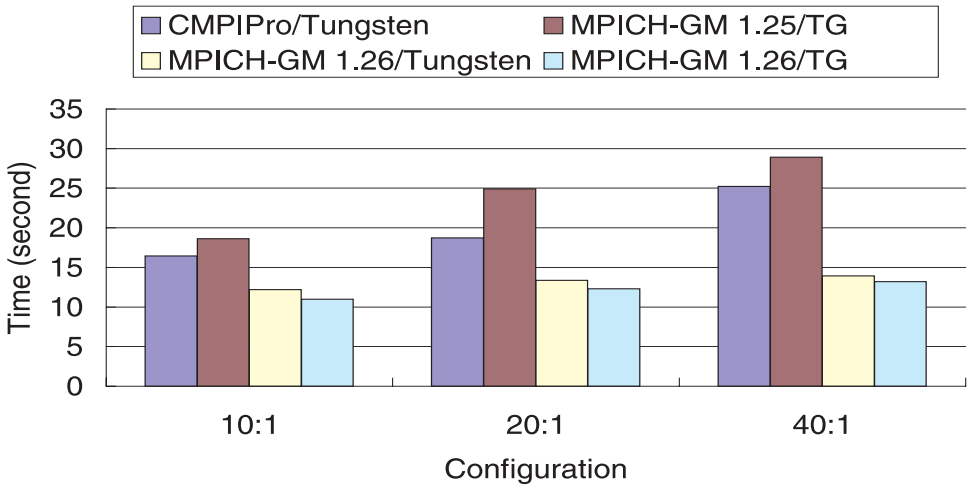


Figure 7.8: Comparison of different reduction algorithms for a fixed per-process checkpoint size of 500 MB.

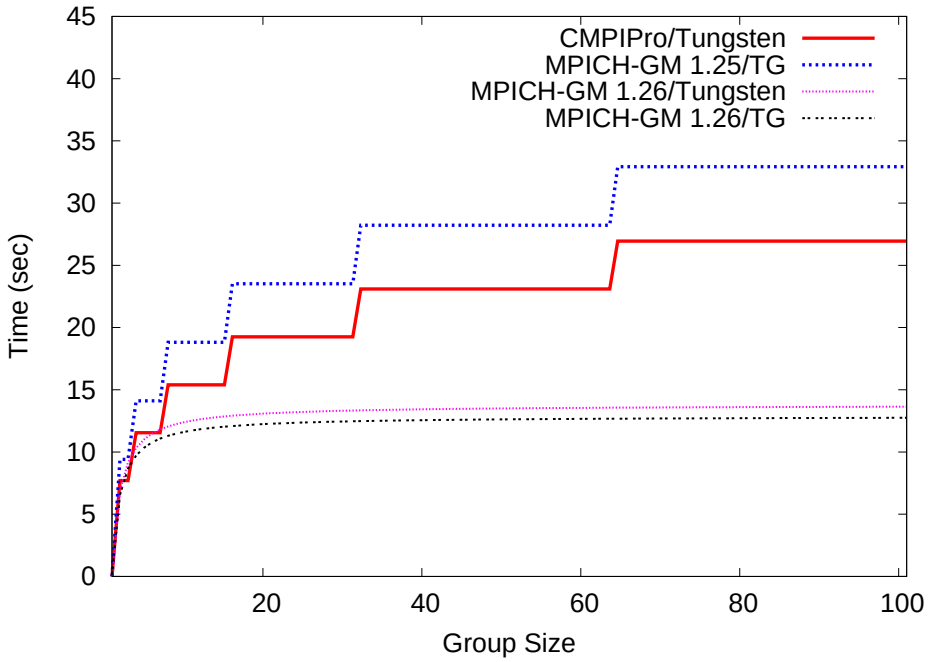


Figure 7.9: Predicted times of different reduction algorithms for a fixed per-process checkpoint size of 500 MB.

where d is per-process checkpoint size in MB and n is group size.

Figure 7.9 plots the performance prediction based on the above equations with $d = 500$. We only predict up to a group size of 100 because our analysis and prediction is based on the assumption that the communication between *any* pair of nodes has the same bandwidth. In a very large system, Thakur [42] has reported that the nearest neighbors can achieve twice more bandwidth than that of two nodes far apart, so actual measurement is still required.

So far we have discussed memory copy time, Reed-Solomon encoding time, and reduction time, we can use these results to establish following formulae about checkpointing time in *one group* case.

If the group has one spare, the time is

$$MemCopyTime(d) + ReductionTime(d, n)$$

If the group has $k > 1$ spares, the time is

$$MemCopyTime(d) + k \cdot (RSEncodingTime(d) + ReductionTime(d, n))$$

We used the formulae to verify the experimental results for *one group* case. The measured times are from the NCSA TeraGrid system with MPICH-GM 1.2.5, in which each process dumps 200 MB of data ($d = 200$). Reed-Solomon encoding rate is assumed to be 207 MB/s (Fully Random data pattern, see §7.3), so $RSEncodingTime(d)$ is $200/207 = 0.97$. Memory copy rate is 1500 MB/s (see §7.2), so $MemCopyTime(d)$ is $200/1500 = 0.13$.

Figure 7.10 presents the comparison. There are four node-spare ratios: 10:1, 20:1, 40:1, and 60:1. In each case the results are normalized against the baseline configuration, which is the measured time of 10:1, 20:1, 40:1, and 60:1, respectively. The error of prediction is 0.2-4.3 percent, with an average of 1.6 percent.

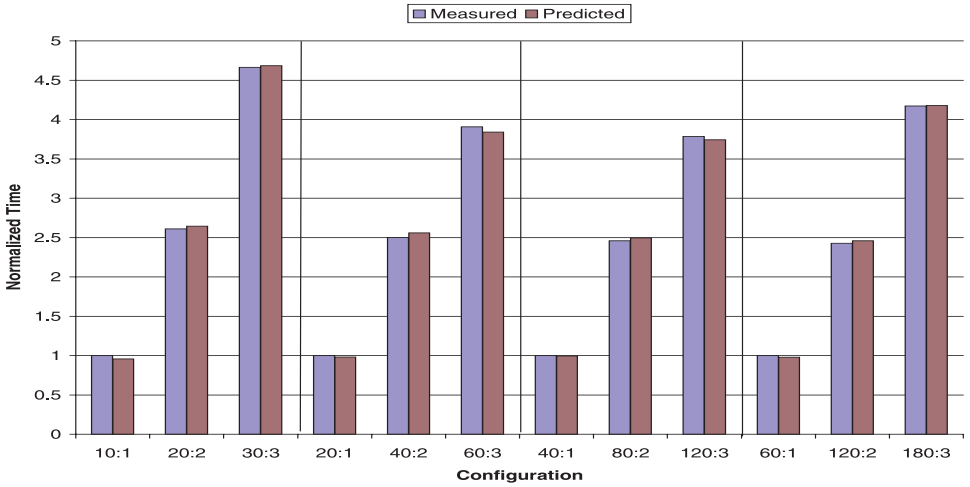


Figure 7.10: Measured and predicted time for multiple spares per group cases.

7.5 Maximum Group Completion Time

Ideally, diskless checkpointing is scalable exactly because the time is constant regardless of number of groups. However, during the experiments, we observed that if there are more groups, the checkpoint completion time of the DLCKPT system preceived at the user application increases. After investigation, we found this is because not all groups complete at exactly the same time, and the reported result is the time the last group that completes. Figure 7.11 shows the completion times of forty-five 10:1 groups on the NCSA TeraGrid system with MPICH-GM version 1.2.5.

To characterize this phenomenon and investigate its growth with respect to the number of groups, we adopt a probability model as follows. Let random variables $X_1, X_2, \ldots, X_n$ denote the completion time of group $1, 2, \ldots, n$, respectively. To simplify, we assume X_i are independent and identically distributed. Then the maximum completion time is defined as $Y = \max(X_1, X_2, \ldots, X_n)$. Y is usually called the n-th order statistic [56]. Order statistics play an important role in reliability analysis. For example, a system made of components structured in a parallel fashion fails when the last good component fails. A branch of statistics called extreme value theory deals with such rare but extreme situations.

Let $f(x)$ and $F(x)$ be the probability density function and cumulative distribution function of

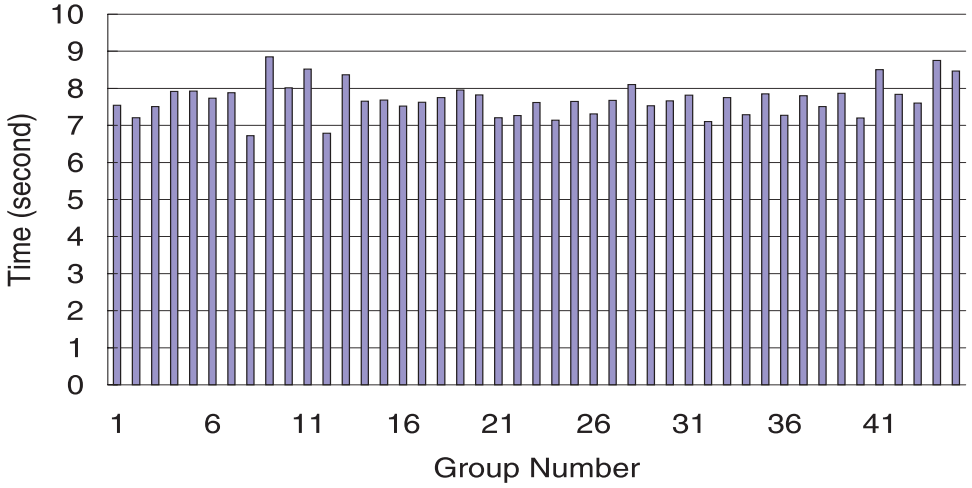


Figure 7.11: Completion time of individual groups. The per-process checkpoint size is 200 MB.

X_i, respectively. Then the mean of Y is

$$E[Y] = n \int_{-\infty}^{\infty} x f(x) [F(x)]^{n-1} \, dx$$

We further assume that X_i's have the *standard* normal distribution (the general case follows easily), that is, $f(x) = (2\pi)^{-0.5} \exp(-0.5x^2)$ and $F(x) = \int_{-\infty}^{x} f(t) \, dt$. Then the limiting distribution (i.e. $n \to \infty$) of Y is called Gumbel distribution, which has a finite mean. This suggests that $E[Y]$ is bounded and does not grow infinitely. Because Y converges to a Gumbel distribution rather slowly, we calculate $E[Y]$ for small n's and list them in Table 7.1. If X_i's have normal distribution with mean μ and standard deviation σ, $E[Y]$ can be directly calculated from Table 7.1 [56]. For example, if $n = 100$, then $E[Y] = \mu + 2.51\sigma$.

To verify this model, we use the experimental result of 10:1 configuration on the NCSA TeraGrid system with MPICH-GM version 1.2.5. We compute the mean and the standard deviation of group completion times and use them to predict the maximum completion times. Figure 7.12 shows the comparison of measured values against the model predictions. The average prediction error is less than one percent.

n	$E[Y]$
5	1.17
10	1.54
50	2.25
100	2.51
500	3.04
1000	3.24
5000	3.68
10000	3.85

Table 7.1: Expectation of the maximum of n i.i.d. standard normal random variables.

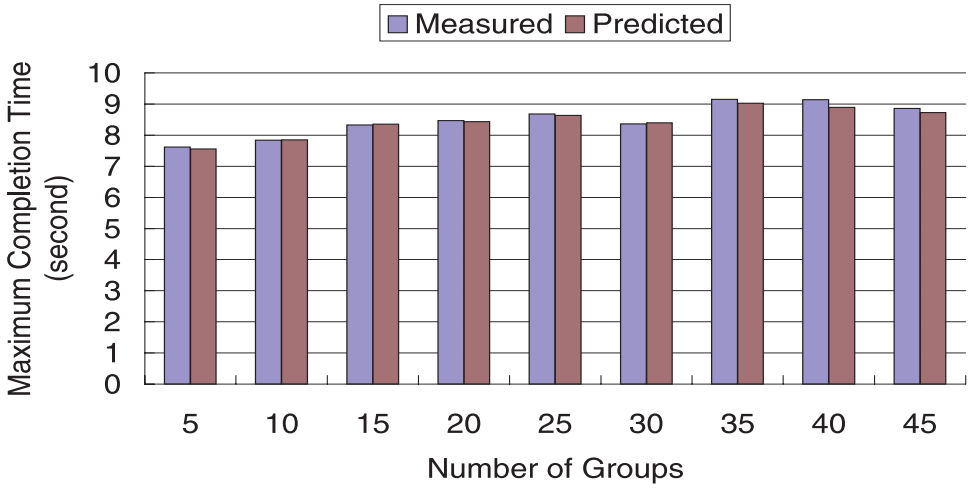


Figure 7.12: Measured and predicted maximum group completion times.

7.6 Summary

We have developed a performance model of the DLCKPT system and derived the model parameters from the experimental results. In this model, we mainly considered the memory copy time, the Reed-Solomon encoding performance and its susceptibility to different data patterns and compiler optimization switches, and the performance of different implementations of `MPI_Reduce()` function. To analyze the scalability of the DLCKPT system, we also investigated the maximum completion times of multiple groups and characterized it with a probability model. Overall, we found our performance model matches closely to the experimental results.

Chapter 8

Performability Modeling

Previously we have experimentally demonstrated the great performance and scalability of diskless checkpointing. However, we have not yet shown the ultimate goal: how diskless checkpointing can help reduce time-to-solution of applications running on large systems. Although diskless checkpointing is fast, its Achilles' heel is that when the number of (hard) failures exceeds available spares in any group. In such a case, which we call **catastrophic failure**, diskless checkpointing simply cannot recover from any further failure.

On the other hand, the user has the liberty of assigning more spares to each group at the cost of slower checkpointing. This poses two interesting questions: based on resources at one's discretion (e.g. total number of spares) how can the user choose a spare assignment strategy to make the application survive as long as possible, and with this strategy what is the overhead to time-to-solution?

For example, given a system of 20 compute nodes and 2 spare nodes, as shown in Figure 6.1, we can create one group with two spares or two groups with one spare each. The former configuration can survive two *arbitrary* failures but also have slower checkpointing, while the latter is contrary.

Collectively the two questions are called the performability (performance and reliability) problem. In this chapter, we build an analytical model to address the performability of diskless checkpointing. As a technology projection, we will evaluate the model numerically and verify it by

simulation in the next chapter.

Before introducing our model, we would like to mention some related performability analysis work. While none of these work characterizes diskless checkpointing exactly the same way as ours does, many of our model assumptions do originate from them. One of the earliest is Young's approximation formula for the optimal checkpoint interval, with the assumption that inter-failure times are exponentially distributed [57]. Gelenbe [58] extened Young's work by allowing arbitrary inter-failure distributions, but the precise result has no closed-form solutions. Daly [59] proposed a modification of Young's model that characterizes a large system composed of individual nodes. Wang *et al* [60] presented a model which further considers the checkpoint protocol overhead and correlated failures. Unlike the previous works which are analytical models, Wang's model is presented in the form of Stochastic Activitity Networks, a variant of Petri nets. Thus, the model can express quite complicate failure relationship among nodes, but no simple formula can be derived from it.

8.1 Model Description

For mathematical convenience we have the following premises in our model:

- Like sPPM and Sweep3D in §6.4, the execution of an application is characterized by the Bulk Synchronous Parallel (BSP) model. Each superstep, or more commonly called, a phase, is composed of Production, during which the application is doing useful work, Checkpoint, and possible Restarts.
- All groups are of the same size and have the same number of spares.
- The failure mode of nodes is fail-stop. Failed nodes do not self-repair nor induce correlated failures of other nodes. There are no transient bit errors in memory or bus.
- Failure detection is accurate and instant. The system triggers a recovery immediately after a failure. To simplify analysis, we do not distinguish between a spare failure or a compute node

failure.

- Recovery is always successful so long as the necessary redundancy codes exist.
- The inter-failure times have an exponential distribution.

We parametrize our model by the following set of variables:

- $\boldsymbol{\lambda}$: Failure rate of a node.
- $\boldsymbol{n}$: Total number of compute nodes.
- $\boldsymbol{m}$: Total number of spare nodes.
- $\boldsymbol{R}$: Number of spares per group.
- $\boldsymbol{N}$: Number of phases of the application execution.
- $\boldsymbol{c}$: Production duration, i.e. the checkpoint interval.

The following parameters are derived from above parameters to aid analysis:

- $\boldsymbol{s}$: Group size, which is $R \cdot \frac{n+m}{m}$.

  The number of groups is $\frac{m+n}{s} = \frac{m}{R}$.
- $\boldsymbol{C}$: Checkpoint time, which depends on many factors as we analyzed in the previous chapter.
- $\boldsymbol{Rs}$: Restart time.
- $\boldsymbol{Rl(t)}$: Reliability function. It is the probability that no failure occurs within t time. Because the system size is $(n+m)$, the system failure rate is $(n+m)\lambda$. Then $Rl(t)$ is equal $e^{-(n+m)\lambda t}$.

Our objective is to first derive the following quantities:

- $\boldsymbol{S(k)}$: Given k failures, the conditional probability that no two failures co-locate within the same group. This is for one spare per group case. Multiple spares per group case will be discussed later.

- $\mathbf{PP(N, k)}$: Probability of k failures in N phases.
- $\mathbf{TT(N, k)}$: The average time-to-solution for N phases of execution, given there are k failures.

And with above quantities, we can calculate the two performability metrics:

- Probability of successful execution of an N-phase execution, which is

$$P(\text{successful execution}) = \sum_{k=0}^{m} S(k) \cdot \text{PP}(N, k)$$

- The average time-to-solution for N phases of execution $\mathbf{TT(N)}$. This assumes the execution is successful, which means there are at most m failures/restarts during the execution. It is given by

$$\text{TT}(N) = \sum_{k=0}^{m} \frac{S(k) \cdot \text{PP}(N, k)}{P(\text{successful execution})} \cdot \text{TT}(N, k)$$

- Checkpoint and restart overhead . We need to define the baseline time-to-solution $\text{IT}(N)$ for N phases of execution. It is the time of an ideal execution free of checkpoints and restarts and is equal Nc. Then the overhead is simply given by:

$$\frac{\text{TT}(N) - \text{IT}(N)}{\text{IT}(N)}$$

8.2 Single Spare Per Group

Initially we study the simple case in which each group has one spare ($R = 1$), and no two failures can co-locate in the same group.

We first derive $S(k)$. The number of ways to place k failures in $m + n$ nodes is $\binom{m+n}{k}$. Beause there are m groups and no two failures can co-locate within the same group, then the number of ways is $\binom{m}{k}s^k$: we first pick k out of m groups, and within each of the chosen groups, we pick 1 out of s nodes and designate it as a failed node. Thus, $S(k) = \binom{m}{k}s^k \Big/ \binom{n+m}{k}$.

Next, we compute an approximation for $\text{PP}(N, k)$. *This approximation assumes the system failure rate is always constant ($= (n + m)\lambda$) regardless of any node loss, albeit in reality the*

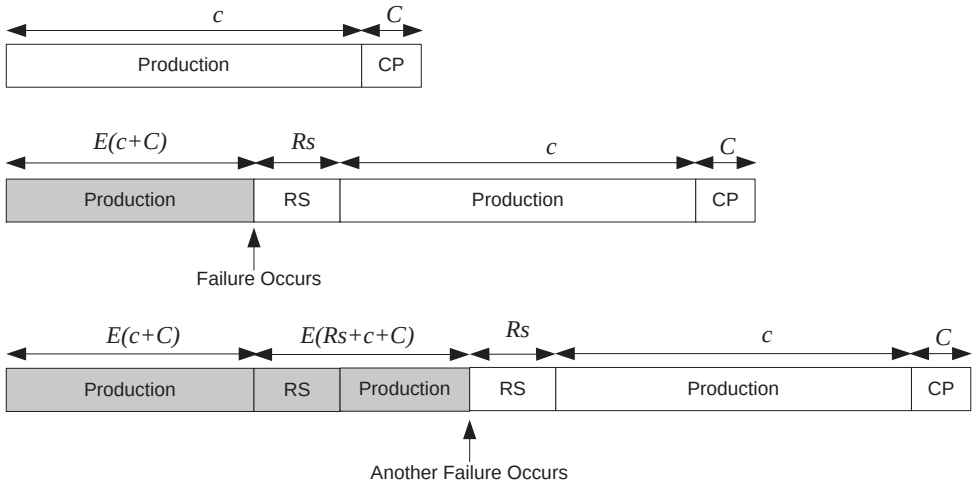


Figure 8.1: Failure scenario in a single-phase execution. CP is checkpoint and RS is restart. The top timeline is failure-free execution, the middle one with one failure, and the bottom one with two failures. The shaded part is the corrupted execution. Its duration $E(c+C)$ and $E(Rs+c+C)$ are counted as the overhead induced by failures.

number of nodes decreases and so does the system failure rate. We made this approximation to simplify analysis. In §9.6, we will verify this approximated model by precise simulations.

To find $\mathrm{PP}(N,k)$, we define two helper functions: the unreliability function $\overline{Rl}(t) = 1 - Rl(t)$, which is the probability of time-to-failure being less than t, and $\boldsymbol{F(k)}$, the probability of k failures in 1 phase:

$$F(k) = \begin{cases} Rl(c+C) & ,k=0 \\ \overline{Rl}(c+C) \cdot \left[\overline{Rl}(c+C+Rs)\right]^{k-1} \cdot Rl(c+C+Rs) & ,k>0 \end{cases}$$

Figure 8.1 illustrates some possible scenarios of one phase execution. Because failures can strike anytime, the chance of a failure-free execution for $c+C$ time is $Rl(c+C)$, and chance of one failure ($k=1$) is $\overline{Rl}(c+C) \cdot Rl(c+C+Rs)$. If more failures occur, then they must occur during a Restart-Production-Checkpoint cycle. Then $\mathrm{PP}(N,k)$ can be defined recursively as:

$$\mathrm{PP}(N,k) = \begin{cases} F(k) & ,N=1 \\ \sum_{i=0}^{k} F(i) \cdot \mathrm{PP}(N-1,k-i) & ,N>1 \end{cases}$$

$\mathrm{PP}(N,k)$ is explained as follows. For an N-phase execution, we divide it into the prior $(N-1)$

phases and the most recent phase. Suppose there are k failures within N phases, then the most recent phase has i failures and the prior $(N-1)$ phases have $(k-i)$ failures, for i runs from 0 to k. For each i, such a situation occurs with probability $F(i) \cdot \mathrm{PP}(N-1, k-i)$.

Finally, we derive $\mathrm{TT}(N,k)$, which is time-to-solution given the execution is successful. As shown in Figure 8.1, each failure induces an overhead which is the wasted time. So given a failure occurs within t time, the average overhead $\boldsymbol{E(t)}$, i.e. time-to-failure, is

$$E(t) = \int_0^t x \cdot \frac{(n+m)\lambda e^{-(n+m)\lambda x}}{\overline{Rl}(t)} dx \; = \frac{1}{(n+m)\lambda} - \frac{te^{-(n+m)\lambda t}}{\overline{Rl}(t)}$$

Given there are k failures within 1 phase, the expected time $\mathbf{PT(}\boldsymbol{k}\mathbf{)}$ is:

$$\mathrm{PT}(k) = \begin{cases} c + C & , k = 0 \\ E(c+C) + (k-1) \cdot E(Rs + c + C) + \Big[Rs + c + C \Big] & , k > 0 \end{cases}$$

To analyze the case for N phases, we again divide the execution into the prior $(N-1)$ phases and the most recent phase. Given there are k failures within N phases, the most recent phase has i failures and the prior $(N-1)$ phases have $(k-i)$ failures, for $0 \leq i \leq k$. For each i, the time is $\mathrm{PT}(i) + \mathrm{TT}(N-1, k-i)$, with probability $\frac{F(i) \cdot \mathrm{PP}(N-1,k-i)}{\mathrm{PP}(N,k)}$ (recall here is the conditional probability, so we need the denominator $\mathrm{PP}(N,k)$.) Combining all of these, we have the average time-to-solution of an N-phase execution given there are k failures:

$$\mathrm{TT}(N,k) = \begin{cases} N \cdot \mathrm{PT}(0) & , k = 0 \\ \sum_{i=0}^{k} \frac{F(i) \cdot \mathrm{PP}(N-1,k-i)}{\mathrm{PP}(N,k)} \cdot \Big[\mathrm{PT}(i) + \mathrm{TT}(N-1, k-i) \Big] & , k > 0 \end{cases}$$

8.3 Multiple Spares Per Group

The analysis for multiple spares case is similar, except that the $S(k)$ function must include the number of spares per group R as a parameter.

Define $\boldsymbol{S(k,R)}$ to be the conditional probability that no more than R failures co-locate in any group, given there are k failures.

To calculate $S(k,R)$ we need to count the number of ways to assign k failures to $\frac{m}{R}$ groups with a limitation that no group has more than R failures. This number can be conveniently found

by using the generating function method [61]. It is equal to the coefficient of x^k, i.e. a_k, in the following generating function:

$$\Big[\binom{s}{0} + \binom{s}{1}x + \cdots + \binom{s}{R}x^R\Big]^{\frac{m}{R}} = \sum_{k=0}^{m} a_k x^k \tag{8.1}$$

First we use this equation to verify the $R = 1$ case we derived earlier. Equation 8.1 becomes

$$\Big[1 + sx\Big]^m = \sum_{k=0}^{m} \binom{m}{k}(sx)^k$$

The right-hand-side is from the binomial theorem. So we have $a_k = \binom{m}{k}s^k$, which is exactly the same as the expression we found in §9.6.

We give a brief explanation of Equation 8.1. For our analysis, $A = \binom{s}{0} + \binom{s}{1}x + \cdots + \binom{s}{R}x^R$ encodes the all possible non-catastrophic failure modes in a group of s nodes: each term $\binom{s}{k}x^k$ embeds two pieces of information: k the number of failures in this group and $\binom{s}{k}$ the number of ways to assign k failures to these nodes.

A simple example which motivates the use of generating functions is as follows. Consider a system of three groups of nodes. Group I has 2 nodes and can survive 1 failure. Group II has 4 nodes and can survive 2 failures. Group III has 6 nodes and can survive 3 failures. All nodes are distinct. The system can survive if no group has failures more than it can tolerate. How many ways are there for 5 failures to occur, yet the system survives? Using the generation function, we can write

$$\Big[\binom{2}{0} + \binom{2}{1}x\Big]\Big[\binom{4}{0} + \binom{4}{1}x + \binom{4}{2}x^2\Big]\Big[\binom{6}{0} + \binom{6}{1}x + \binom{6}{2}x^2 + \binom{6}{3}x^3\Big] \tag{8.2}$$

It can be verified that the number of ways to be sought is the coefficient of x^5 in Equation 8.2, which is $\binom{2}{1}\binom{4}{2}\binom{6}{2} + \binom{2}{1}\binom{4}{1}\binom{6}{3} + \binom{2}{0}\binom{4}{2}\binom{6}{3}$. The case for 5 is manageable without using generating functions, but the case for 3 will be complex and generating function is the easiest way to count.

Therefore, in Equation 8.1, the coefficient a_k of x^k in A^j is the number of ways to assign k failures to j groups such that no group has more than R failures. Although the explicit formula for

a_i is very complex when $k > 1$, the generating function method gives a straightforward and quick way to count.

To obtain the coefficient a_k, we need to expand A^j. An efficient way to do this for large j's is to calculate A^2, A^4, A^8, ... until A^n where n is the largest power of 2 that is less than j. For example if $j = 100$, then we calculate and keep the results up to A^{64}. Then we multiply A^{64}, A^{32}, and A^4 together to obtain A^{100}.

8.4 Summary

In diskless checkpointing, if the number of failures in a group exceeds the number of spares in that group, then the application cannot recover from the failure.

Based on the above premise, we have used the probability theory to establish and analyze an analytical model of diskless checkpointing. This model addresses two issues: given a diskless checkpointing configuration (e.g. two spares per every twenty nodes), what is the success rate of application execution if the nodes fail randomly? And what is the associated time overhead due to checkpoints and restarts?

This model can assist the user to choose an appropriate configuration that best fits his needs. In particular, given a fixed number of spares, the user can use this mode to determine how to allocate spares to a group to achieve the maximum success rate of execution or minimize the checkpoint and restart overhead.

Chapter 9

Projections for Large Systems

Despite hardware and time constraints that prevent us from experimentally verifying the performability of diskless checkpointing on large scale systems, we can still provide a technology projection based on the model developed in the last chapter.

In this chapter we use both numerical evaluation and simulation to predict the performability the user can anticipate on large systems. We first describe and evaluate the baseline case in §9.1. We conducted a series of parametric studies and the results are presented in §9.2 through §9.5, which are followed by simulation results in §9.6.

9.1 The Baseline Case

We adopted the following set of parameter values as our baseline case. A summary is in Table 9.1.

- λ: **Node failure rate**, which is 1/(43,800 hours). This is equal to an MTTF of five years. We choose this value because most computer vendors provide warranty no longer than this period, and the life-cycle of supercomputers is usually three to five years. Five years of MTTF can be translated as 0.999977 (approximately five nine's) of one-hour reliability.

- n: **Number of compute nodes**, which is 5,000. There are several big systems with size of this order of magnitude, such as Thunder (4,096 processors) at the Lawrence Livermore Na-

tional Laboratory and Seaborg (6,656 processors) at the National Energy Research Scientific Computing Center (NERSC).

- m: **Number of spare nodes**, which is 250. This gives a 20:1 node-spare ratio, or 5 percent of redundancy, which we believe is a reasonable assumption for a big system.

- C: **Checkpoint time**. We assume the baseline system to have hardware and software configuration similar to NCSA TeraGrid's. Namely, memory copy rate is 1,500 MB/s (see §7.2), Reed-Solomon encoding rate is 207 MB/s (see §7.3), the MPI library is MPICH-GM version 1.2.5 (see §7.4), the standard deviation of group completion times is 1 second (see §7.5), and a communication overhead between DLCKPT codec module and the user application of 1.5 seconds.

  The checkpoint size per process is 200 MB.

- Rs: **Restart time**. The restart time follows the model derived in § 7.1. We use 2.5 seconds of failure detection time, 3 seconds of process recovery time (measured in experiments with sPPM and Sweep3D in §6.5), and 10 seconds of job launching time (use the technology mentioned in §6.5). We also assume the user application use an extra of one minute to do bootstrapping and reconfiguration (e.g. reinitialize data structures.)

- c: **Production duration** (i.e. checkpoint interval) The user application performs diskless checkpointing after every c time of work. Large parallel programs checkpoint every 1-3 hours [41, 4]. In theory, checkpointing less often will not reduce overhead because more work could be lost in one failure. To determine the checkpoint interval we use Young's formula [57], which gives the optimal checkpoint interval as $c = \sqrt{2MC}$ where C is the checkpoint time defined as above, and M is the system MTBF. In the baseline case, $M = 43,800/5,500 \approx 8$ hours, and C is no greater than one minute = 0.017 hours, so $\boldsymbol{c} \approx 0.5$ hours.

We also define the two metrics as the criteria for assessing the effectiveness of diskless checkpointing:

Parameter	Value	Comments
λ	1/43,800	Node failure rate
n	5,000	Number of compute nodes
m	250	Number of spare nodes
C	(see Table 9.2)	Checkpoint time
Rs	(see Table 9.2)	Restart time
c	0.5	Production duration (hours)

Table 9.1: Summary of parameters of the baseline case.

Spares per group	Checkpoint time C	Restart time Rs
1	0.22	1.48
2	0.47	2.19
3	0.66	2.58
4	0.98	3.22

Table 9.2: Checkpointing and restart times (in minutes) of the baseline case.

- **Reliability**: The number of phases that can be successfully executed at probability 0.9. The greater the better.
- **Overhead**: The fraction of additional execution time induced by checkpoint/recovery. The smaller the better.

Putting these parameter values into the formulae in the last chapter, we calculate the checkpointing and restart times and list them in Table 9.2. Figure 9.2 (c) shows the performability curve of the baseline case. Clearly the longer the application runs, the less likely it can accomplish without encountering a catastrophic failure.

A more detailed comparison of spare and group partitioning schemes is presented in Figure 9.1. We find 2,3, and 4 spares per group schemes can increase reliability (compared to one spare per group) by a factor of 3, 5, and 7, respectively. We also found the overhead increases by 26, 46, and 78 percent, respectively, and in all cases the overhead is in the range of 0.04-0.07. In the next few sections we present the results of sensitivity analysis.

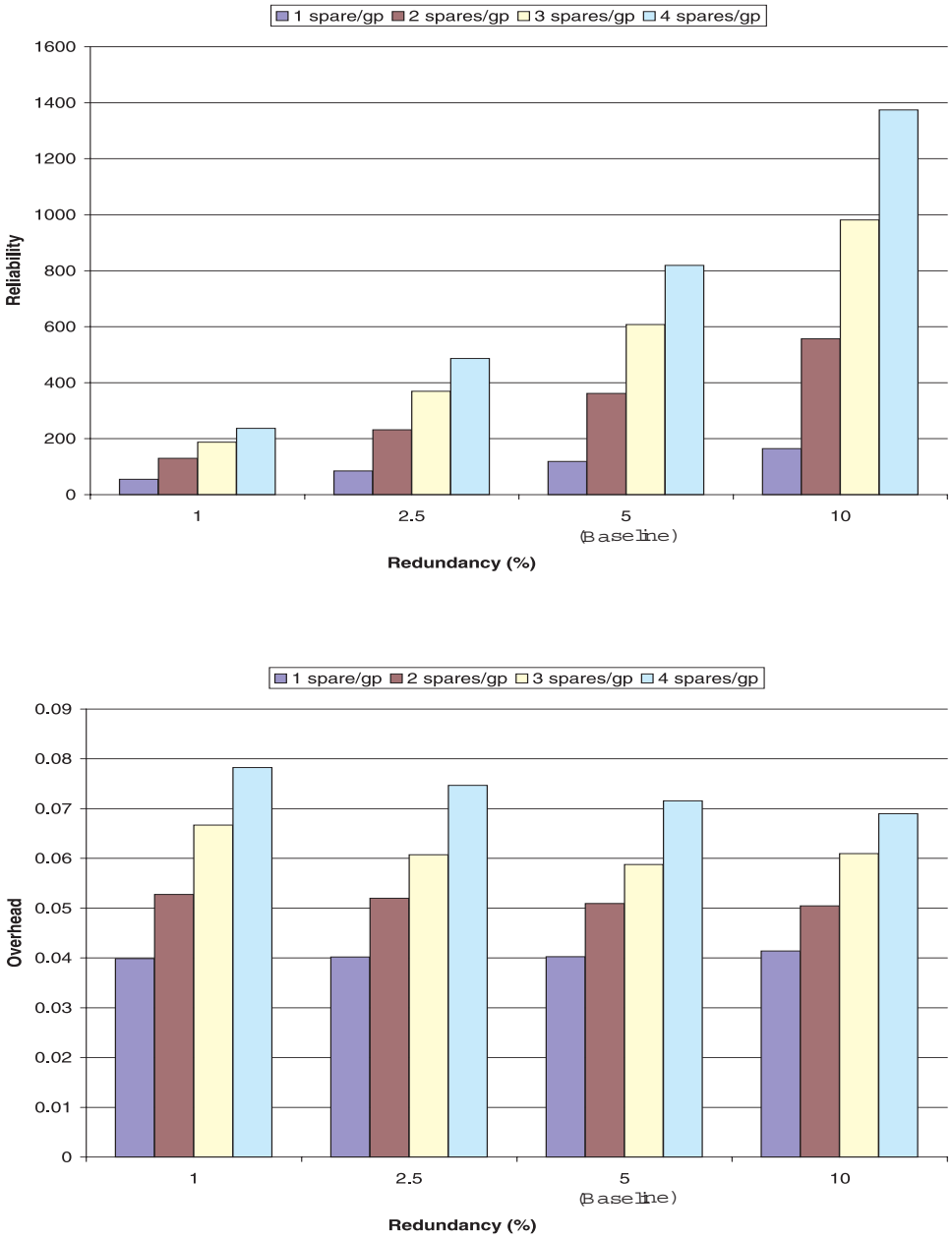


Figure 9.1: Reliability and overhead of a system of 5,000 compute nodes.

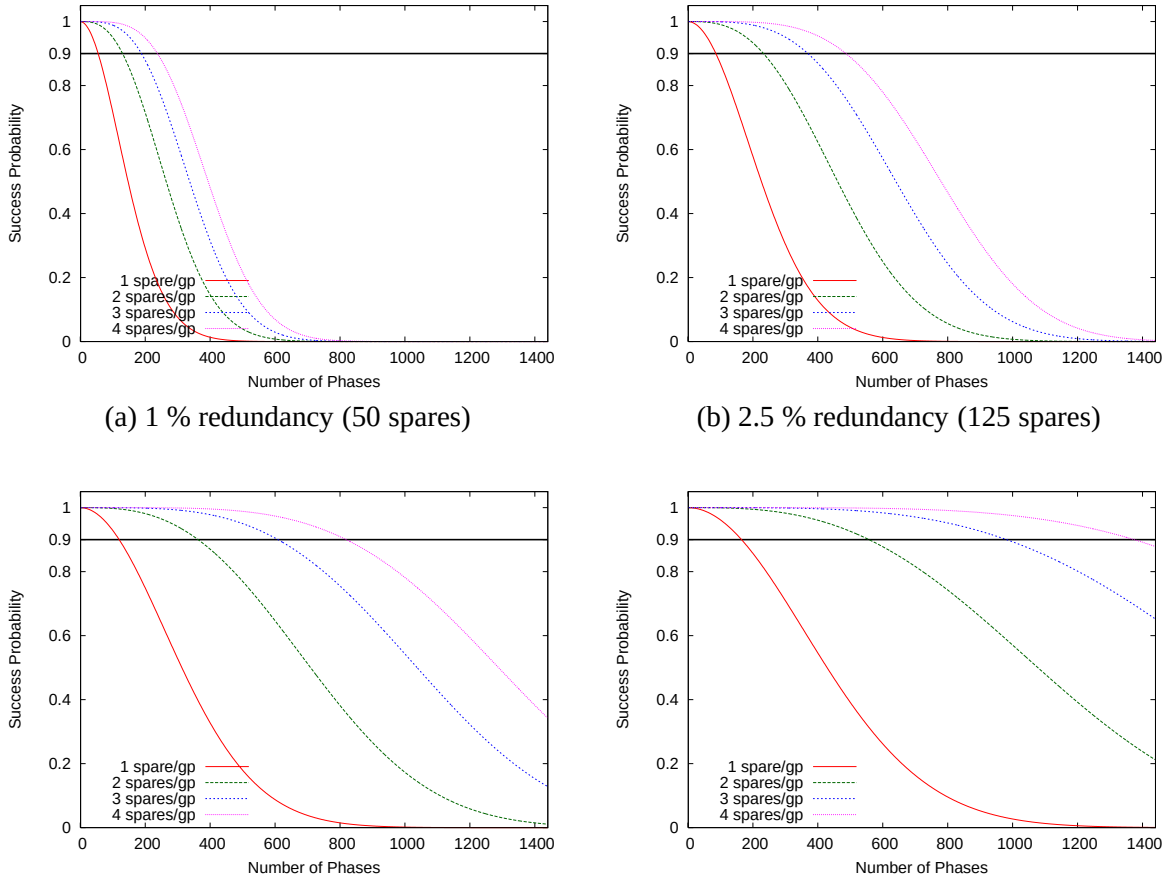
9.2 Sensitivity Analysis: Total Number of Spares

Previously we assumed the system has a 5 percent redundancy. Here we explore other node-spare ratios: 10:1 (10 percent), 40:1 (2.5 percent), and 100:1 (1 percent).

The results are shown in Figure 9.2 and 9.1. We note that the performability curves have similar shapes and only differ in the spread. More spares will undoubtedly improve reliability. This improvement, however, is not proportional to the number of spares.

For example, comparing the baseline to the 10 percent redundancy case, we found that doubling the spares only improves reliability by 1.39-1.68 fold. The diminishing returns of adding more spares are also evident if we compare the baseline to the 2.5 percent redundancy case.

For overhead we found it drops slightly in higher redundancy case. This is because the group size is smaller, so the reduction times get shorter.



(a) 1 % redundancy (50 spares)

(b) 2.5 % redundancy (125 spares)

(c) 5 % redundancy (250 spares)

(d) 10 % redundancy (500 spares)

Figure 9.2: Performability curves of a system of 5,000 compute nodes. The duration of time axis is 1,440 phases, which are roughly equal 30 days of execution (each phase is 0.5 hours, excluding checkpoint/restart.)

9.3 Sensitivity Analysis: System Size

We also considered systems of different compute node counts: 10,000 and 15,000. The results are summarized in Figure 9.3.

First, we noted that the plots of different system sizes have resembling shapes, that is, the relative magnitude are similar. The main difference, of course, is the absolute magnitude. In the 10,000 nodes case, the reliability is in the range of 37-1100, and in the 15,000 case, the the reliability reduces to 29-1000.

For overhead, we also found that it increases from 0.07-0.12 in 10,000 case to 0.11-0.16 in 15,000 case. The reason of lower reliability and higher overhead is not hard to surmise: the system MTBF becomes shorter (4.4 hours for 10,000 nodes and 2.9 hours for 15,000 nodes), so the system fails often and spends more time in recovery.

If we apply Young's formula to adjust the checkpoint interval according to the system MTBF, then the checkpoint interval will be shorter for larger systems, e.g. 0.28 hours for 15,000 nodes. We found that the overhead is reduced somewhat (from 0.11-0.16 to 0.09-0.13) but the reliability does not change much.

9.4 Sensitivity Analysis: Node Failure Rate

The overall system failure rate is dependent on the node failure rate. In the baseline case we used a node MTTF of 5 years. Here we explore what happens if the node MTTF is 3 or 7 years, or equivalently, the one-hour reliability is 0.999962 or 0.999984, respectively.

Although the difference seems small (≈ 0.002 percent), it has vast impact to system reliability. Figures 9.4 summarizes the results. In the 3 years case, the maximum of reliability is 800, while in the 5 years case, it is close to 2000. We found the reliability is roughly proportional to the single node MTTF. For example, if the node has 7 years of MTTF, then the system is 1.42 more reliable than that of nodes with 5 years of MTTF.

The overhead is as susceptible as reliability. In the 3 years case, the overhead is in the range of

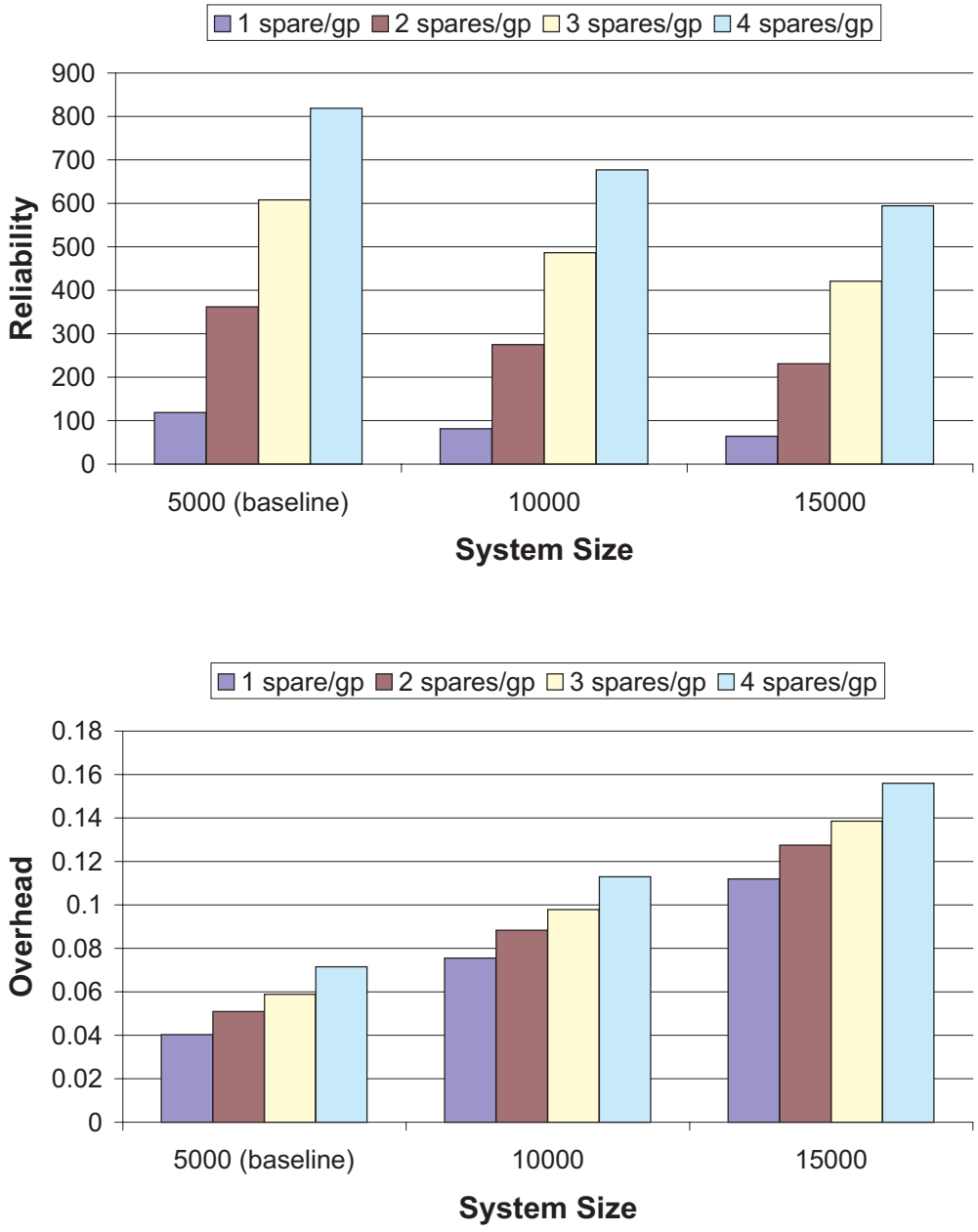


Figure 9.3: Reliability and Overhead of different system sizes.

	MPICH-GM 1.2.5 (baseline)		MPICH-GM 1.2.6 (fast MPI)	
Spares per group	C	Rs	C	Rs
1	0.22	1.48	0.07	1.33
2	0.47	2.19	0.11	1.47
3	0.66	2.58	0.12	1.50
4	0.98	3.22	0.14	1.54

Table 9.3: Checkpoint and restart times (in minutes) of the baseline system and a faster system.

0.06-0.1, while in the 5 years case, it is in the range of 0.03-0.067.

9.5 Sensitivity Analysis: Checkpoint and Restart Times

We used MPICH-GM version 1.2.5 as the communication middleware in the baseline case. As explained in §7.4 the improved algorithm in MPICH-GM version 1.2.6 increases performance by a factor of 2 or more over older versions. In this section, we compare the performability of two systems running on MPICH-GM version 1.2.5 and 1.2.6, respectively. Table 9.3 lists the checkpointing and restart times (in minutes) for a system of 5,000 nodes and 250 spares.

The result is shown in Figure 9.5. It can be readily seen that although the checkpoint and restart times are much shorter, the improvement in reliability is less than 5 percent. On the other hand, the overhead does drop from 0.04-0.07 to 0.03-0.04.

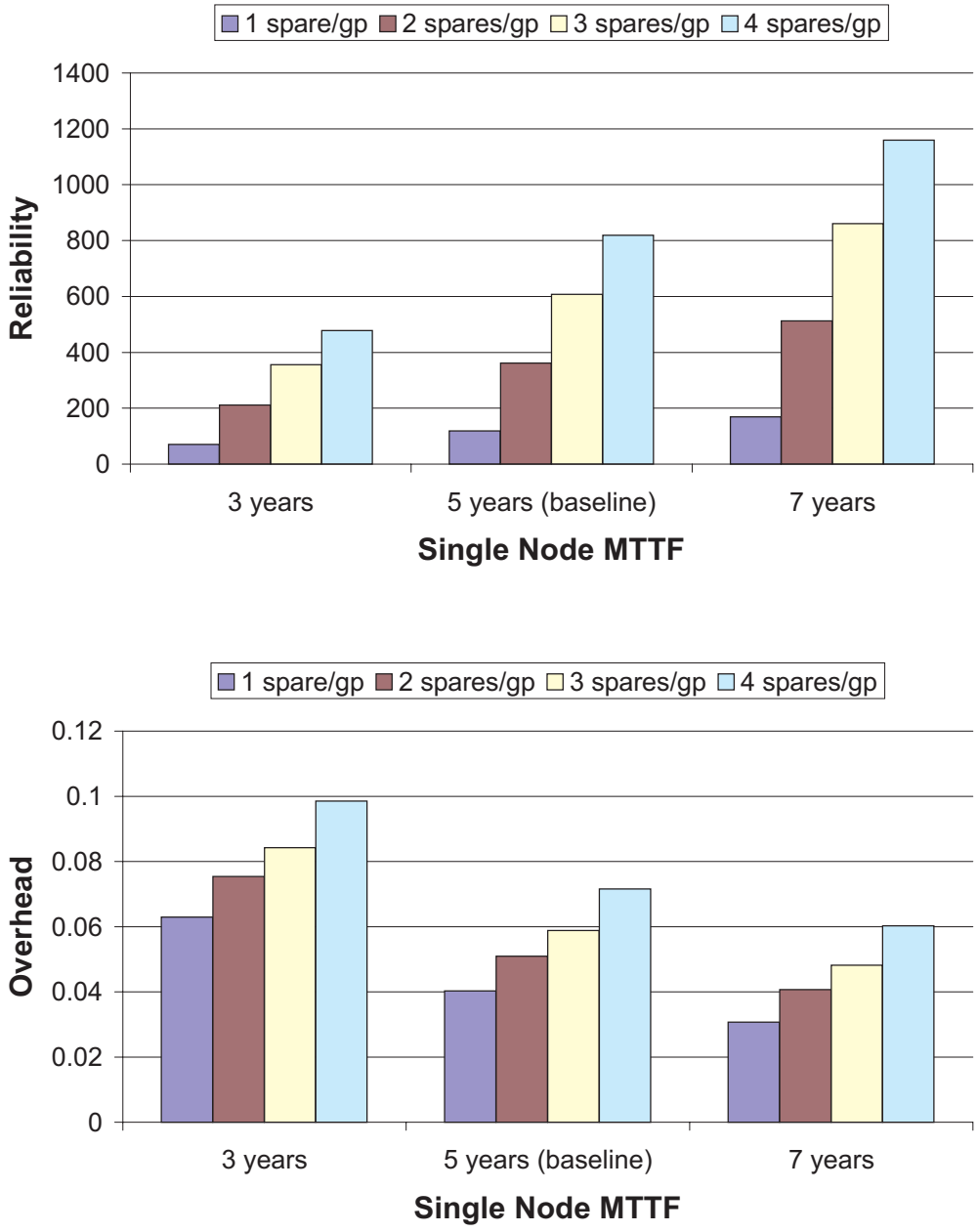


Figure 9.4: Reliability and Overhead of different node failure rates.

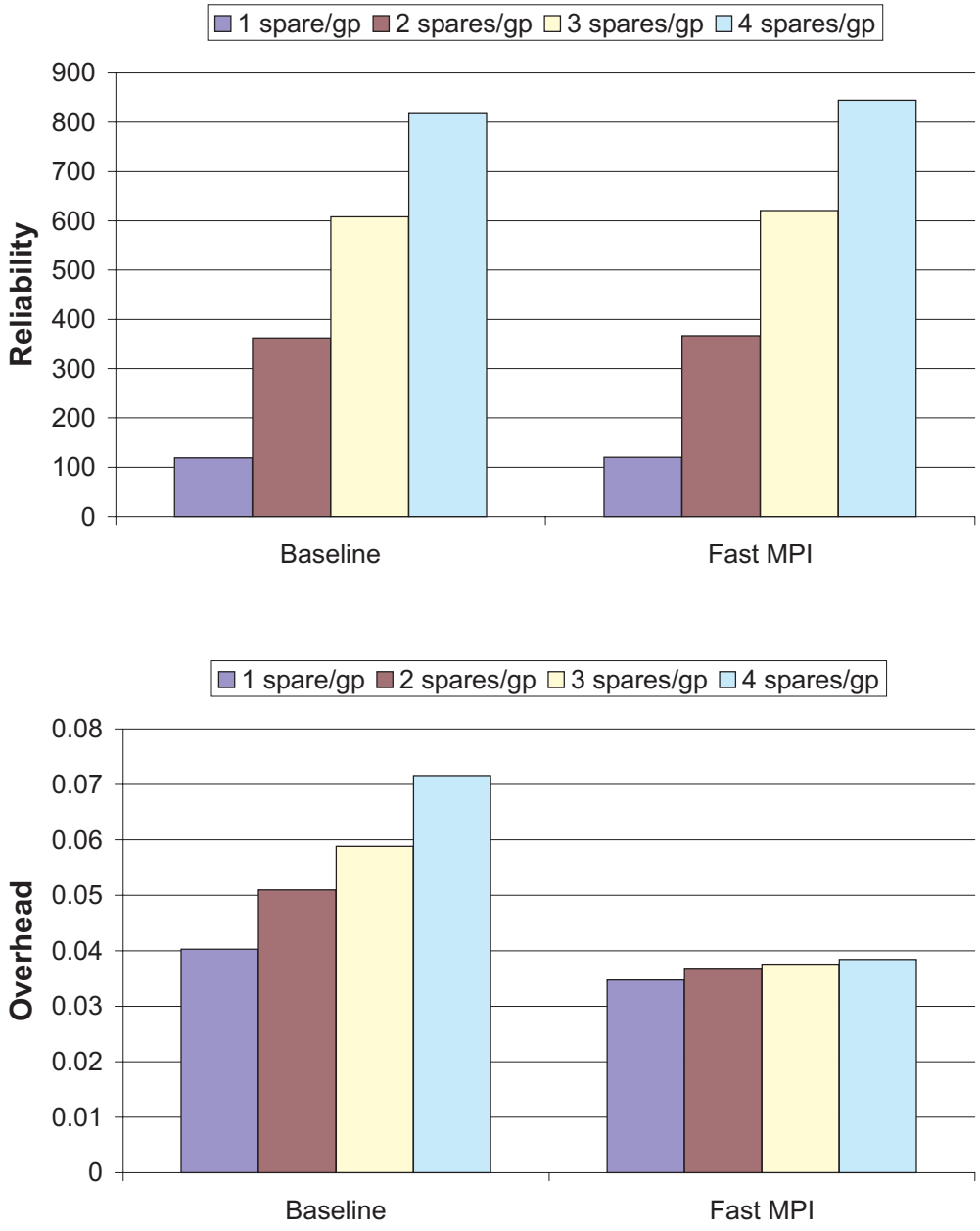


Figure 9.5: Reliability and Overhead of different checkpoint and restart performance.

9.6 Simulation

Our performability model developed in the previous chapter assumes that the system failure rate remains constant no matter how many nodes have been lost (see §9.6). We made this approximation to simplify mathematical analysis. To corroborate this model, we write a simulation program which is based on the same model but the system failure rate varies with respect to the number of remaining nodes.

Each simulation run roughly proceeds as follows. We update the system failure rate when a node is down and use the new failure rate to generate an exponential random variate t, the time to next failure. We calculate the number of elapsed failure-free phases within t and the overhead of checkpoints and restarts. Then we randomly mark a up node as down and check for the catastrophic failure condition. If the condition holds, we label this run as failure. Otherwise, we generate a new time-to-failure and repeat the previous process until the total number of elapsed phases exceeds a pre-defined value k, in which we label the run as success. For each k the above simulation run is repeated thousands of times, and the fraction of successful runs and the average overhead are reported.

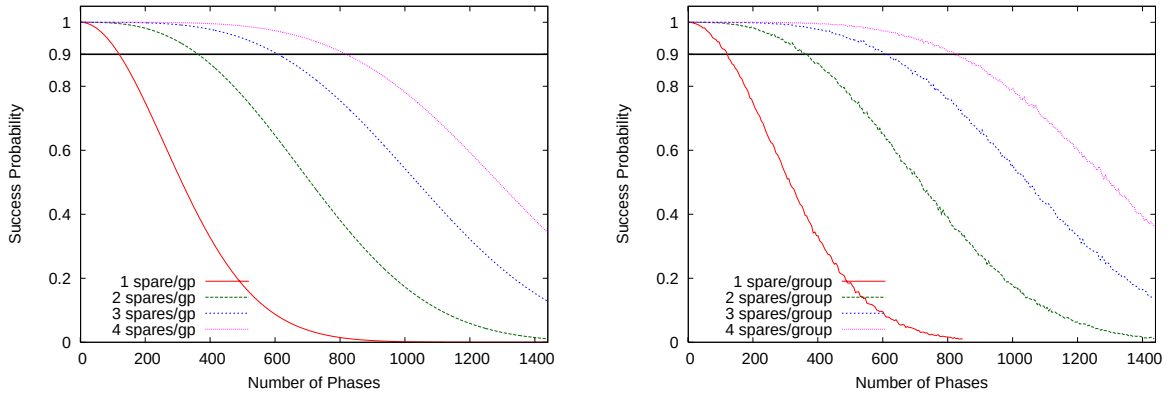
The performability curves of the baseline case are show in Figure 9.6. For other configurations, the simplified model also exhibit quite accurate results, and the error is less than 5 percent.

9.7 Summary

From the evaluation of our performability model we have the following observations:

- If the total number of spares is fixed, the best way to use them is to assign multiple spares per group. Although the group size will be larger, the reliability increases greatly at small overhead increase.

  For example, 4 spares per group is 6.9 times more reliable than 1 spare per group scheme. The overhead increases from 4 to 7.2 percent.



(a) Model (b) Simulation

Figure 9.6: Comparison of model and simulation.

- The reliability is proportional to the total number of spares, although the relationship is not linear. For example, doubling the spares only provides 1.7 times more reliability at best.

- The overhead for larger systems is greater because large systems tend to fail often.

  For example, the overhead for a system of size 5,000 is at most 7 percent but for a system of size 20,000 it is 20 percent.

- The node failure rate has a great impact on system reliability. The system reliability is roughly proportional to the MTBF of a single node. Systems built from nodes with an MTBF of 7 years can increase system reliability by a factor of 1.43 compared to systems built from nodes with an MTBF of 5 years.

- Checkpointing and recovery performance can affect the overhead as much as a factor of 2.

- Although our performability model is a simplified one, we have verified it against simulation based on more precise conditions and found it can indeed characterize the performability of diskless checkpointing with high fidelity,

Chapter 10

Related Work

In this chapter, we survey the related literature on failure data analysis, diskless checkpointing, fault-tolerant MPI, and fault tolerance research for large systems.

10.1 Failure Data Analysis

Although real failure data is not easily accessible, among the few publications it is agreed that software is the major source of unplanned outages. Among hardware outages, disks are the most unreliable component [62, 63, 64], and this fact contributes to the invention and wide adoption of RAID. The seminal work of Gray [65] analyzed 2,000 Tandem fault-tolerant systems over a seven-month period and found that next to preventive maintenance, software problems caused most downtime. The work also pointed out the double failure phenomenon, in which shortly after maintenance comes an unscheduled outage, is not rare. For example, the operator could have typed the wrong command or unplugged the wrong components during maintenance.

Five years after the previous paper, a follow-up availability analysis [66] reported that software quality did not improve much; it still accounts for majority (> 60 percent) of unscheduled downtime. The paper concluded that system software complexity grows to support new peripherals, network protocols, or other customized functionality. As hardware becomes increasingly reliable, software troubles naturally dominate the cause of outages.

In a distributed computing environment, outages can propagate across the network. Wood [62] mentioned the "broadcast storm" phenomenon, in which a great number of random packets were injected into the network. The heavy traffic slowed down the system and caused poor response time. The user's perception of poor response time was the LAN was down. The propagating errors are not necessarily due to hardware problems only. Buggy software, computer viruses, distributed denial-of-service attacks, and even problematic routing algorithms in switches and routers can also cause this kind of troubles. Xu *et al* [67, 68] used Markov chains and clustering analysis study correlated failures in a network of Windows NT workstations. Mutka and Livny [69] and Brevik *et al* [70, 71] studied the inter-failure times of machines in distributed computing environments by statistical distributions and used the results to predict machine availability.

Assorted domain-specific failure data have also been studied: operating systems [72, 73, 74], disk drives [63], human factor [75, 76], impact of workload on failure trend [77], DEC VAX systems [78, 79], Internet services [80], Internet backbones [81, 82] and telephone network [83].

Field failure data of high-performance computing systems is usually for internal circulation and is almost never published in detail. However, there are several talks and reports that shed light on the actual operations of some of the world's most powerful supercomputers. Table 10.1 summarizes the reliability of such systems. In general, these systems are designed for multiple-month, thousand-processor simulations [84].

10.1.1 ACSI White

The ASCI White is a 8,192-processor IBM RS/6000 SP supercomputer with peak performance of 12.28 teraflops. Koch [41] reported that the ASCI White had successfully completed a four-month 2,000-4,000 processor run of a nuclear explosion simulation. Such large jobs usually dump 100-250 GB checkpoint data every hour, and full-time human operators are required to monitor the execution progress.

Each whole-system reboot of the ASCI White system takes 4 hours and preventive maintenance is performed weekly, with separate periods for software and hardware. According to the report,

machine problems occur in every aspect of the system. For example, transient CPU faults generated invalid floating-point numbers, and it took great efforts to spot these bad nodes because they passed standard diagnostic tests and only failed in real programs. Bad optical interconnects led to non-repeatable hard-to-trace link errors which corrupted the computation because these errors could sneak through network host firmware and MPI library without being detected. The storage system was not 100 percent dependable either. The parallel file-system sometimes failed to return I/O error to the user program when the program was dumping restart files. In addition, the archival subsystem's buggy firmware corrupted restart files and made the user program fail to launch.

Seager [85] showed that the reliability of the ASCI White improved over time as MTBF increased steadily from 5 hours in January 2001 to 40 hours in February 2003. Except uncategorized failures, the storage system (both local disks and IBM Serial Disk System) proved to be the main source of hardware problems. The next hardware problems are CPU and third-party hardware. For software, communication libraries and operating systems contributed the most interruptions.

10.1.2 ACSI Q

The ASCI Q system at Los Alamos National Laboratory is a 8,192-processor supercomputer that can provide 20 teraflops computational capability. Morrison [86] reported operations of the ASCI Q system during June 2002 through February 2003. The MTBI is 6.5 hours, and on the average there were 114 unplanned outages per month.

Morrison mentioned that file system is the top priority among the problems to be solved for the ASCI Q system. The troubles include loss of data on local scratch disks, considerable impact of local disk failures on the whole system (many "hung" services require whole machine reboot, which takes 4-8 hours), and occasional unavailability of files.

In addition to the storage subsystem, hardware problems account for 73.6 percent of node outages, with CPU and memory modules being responsible for over 96 percent of all hardware faults (CPU is 62.5 percent and memory is 33.6 percent.) Network adaptors or system boards seldom fail.

10.1.3 Other Large Systems

Levine [87] described the failure statistics of Pittsburgh Supercomputing Center's supercomputer Lemieux: the MTBI during the period from April 2002 to February 2003 is 9.7 hours, shorter than predicted 12 hours. The availability is 98.33 percent[1] during mid-November 2002 to early February 2003.

The National Energy Research Scientific Computing Center (NERSC) houses several supercomputers and their operations are summarized in NERSC's annual self-evaluation reports and NERSC's website [88, 89]. During the period from August 2002 to July 2003, NERSC's largest supercomputer Seaborg reached 98.74 percent scheduled availability, 14 days MTBI, and 3.3 hours MTTR. Storage and file servers had similar availability. Two-thirds of Seaborg's outages and over 85 percent of storage system's outages are due to software.

In the realm of very large scale Internet services, Google operates its search engine in five geographically dispersed data centers consisting of over 15,000 cheap PCs and disks storing petabytes of data [90]. Hennessy and Patterson [64] reported that Google's biggest source of failure is software, which is mostly fixed by reboot and roughly 20 machine are rebooted every day. Hardware has about 1/10th the failures of software. Ninety-five percent of hardware problems are disks and DRAMs, and the remaining five percent are due to problems with the motherboard, power supply, connectors, and so on. Two to three percent of the PCs replaced annually. Transient bit-flip errors in DRAM and during data transfer are also detected. Disks usually experience a performance degradation before crash.

10.2 Diskless Checkpointing

The idea of replacing rotating magnetic media by semiconductor memory as the preferred storage can be traced back to 1980s, but not until recently does it become a reality thanks to the declining price of memory and the broad adoption of compact mobile devices such as digital cameras, PDAs,

[1] Estimated from "Time Lost to Unscheduled Events" divided by weekly node-hours product.

Computer (Site)	**Configuration**	**Reliability**
ASCI Q (LANL)	8,192 1.25GHz Alpha EV-68 processors. Quadrics interconnect. Peak: 20 teraflops	MTBI: 6.5 hr (06/2002-02/2003) 114 unplanned outages/month. Main HW outage sources: storage, CPU, and memory
ASCI White (LLNL)	8,192 375MHz Power3 processors. IBM SP/2 switch. Peak: 12 teraflops	MTBF: 5 hr (01/2001), 40 hr (02/2003) Main HW outage sources: storage, CPU, and third-party hardware
Lemieux (PSC)	3,016 1GHz Alpha EV-68 processors. Quadrics interconnect. Peak: 6 teraflops.	MTBI: 9.7 hr (04/2002-02/2003) Availability: 98.33% (11/2002-02/2003)
Seaborg (NERSC)	6,656 375MHz Power3 processors. IBM SP/2 switch. Peak: 10 teraflops	MTBI: 14 days MTTR: 3.3 hr Scheduled availability: 98.74% (08/2002-07/2003) Main outage sources: software
Google	$\sim$ 15,000 Pentium 3/4 processors	20 machine reboots/day. 2-3% machines replaced annually. Main HW outage sources: storage and memory

Table 10.1: Summary of reliability of large systems

and Flash memory drives.

I/O has always been critical to database performance, prompting researchers in mid-1980s to consider fitting the entire database into main memory [91]. However, disks cannot be totally eliminated and in-memory data or transaction logs must still be checkpointed to magnetic media. Another drawback was that, at that time, memory was still at a premium; only a handful of applications in telecommunications and real-time systems could afford enough memory to store all their data.

In the context of operating systems, much work has been done on in-memory file systems. Non-volatile RAM (NVRAM) has been employed to alleviate write traffic and provide reliability [92, 93, 94, 95]. Traditional write cache still suffers from high latency, since in-cache newly-written data (dirty data) must still be written to disk (flushing) to ensure persistence. If the write cache is implemented in NVRAM, flushing can be postponed until the cache is full, and dirty data can be preserved across reboots or crashes.

Portable devices also requires in-memory file systems. For example, Flash memory is one kind of NVRAM that has become ubiquitous in hand-held computing devices. The access latency of Flash memory is slower than DRAM, especially for writes because the content must be erased before new data can be stored. The erasure process must be performed block-wise (64-128 KB per block) and takes about 0.6-0.8 seconds per block. As a result, Flash-based file systems need new algorithms and techniques for block replacement. Example Flash file systems include [93, 96].

RAM disk is an idea similar to in-memory file systems and is supported by many operating systems [97]. A RAM disk is a pre-allocated portion of main memory that mimics a hard drive. It is mostly implemented as a memory-resident program or a kernel driver. RAM disks are normally used to store short-lived data such as scratch or temporary files. Unlike a disk buffer, RAM disks give users complete control over the content they wish to store.

Diskless checkpointing can be thought as a group of RAID-enhanced RAM disks. The idea could be traced back to Silva and Plank's work [98, 99]. Silva [98] implemented "checkpoint mirroring" on Transputer networks and Plank [99] proposed using redundancy codes to improve recoverability. Checkpoint mirroring is simply storing a node's checkpoint in a neighboring node's mem-

ory. It can survive multiple failures so long as failures do not happen on adjacent nodes at the same time. Chiueh [100] experimented both checkpoint mirroring and parity on the massively parallel computer Maspar DECmpp 12000 and found that mirroring is ten times faster than parity scheme, at expense of double memory space. Plank extended the parity redundancy to Reed-Solomon codes to allow multiple, *arbitrary* simultaneous failures in [35].

More recently there is a revival of interest in diskless checkpointing. Chen *et al* [101] used checkpoint mirroring together with a fault tolerant MPI library to provide reliability to parallel programs. Zheng [102] modified checkpoint mirroring by storing the checkpoint on *two* other nodes, hence strengthening survivability of multiple failures. In [103] Chen also studied a new class of redundancy codes. Most redundancy codes treat data as a semantic-less stream of bits. Chen proposed redundancy codes defined over real or complex numbers. Of course, due to the limit of floating-point numbers, lost data cannot be reconstructed exactly, but Chen's method can recover lost data with pretty good approximation.

Our main contribution is to extend Plank's work by making diskless checkpointing scalable on large systems through partitioning nodes into groups. We also develop and evaluate a performability model to determine the group size and choice of redundancy codes that balance the trade-off between overhead and reliability.

10.3 Fault Tolerant MPI and Checkpointing Libraries

There is a trend in the high-performance computing community of providing fault tolerance in MPI programs. One class of fault tolerant MPI libraries focuses more on dynamic configurability and robustness of run-time execution, such as Harness [104], MPI/FT [105], MPI-FT [106], and Starfish [107]. Another class of fault tolerant MPI libraries couples tightly to the checkpoint/restart mechanism being used. These are more like checkpointing libraries for distributed systems. Examples include CoCheck [108], MPICH-V [109, 110], LAM/MPI [111], MPI/Pro [112, 113]. All of these depend on certain system-level transparent checkpoint libraries such as Libckpt [114], Condor [115] or BLCR (Berkeley Lab's Linux Checkpoint/Restart) [116]. Some mainframe vendors such as IBM,

SGI, and Cray also implement parallel job checkpoint/restart capability in their operating systems or execution environments.

FT-MPI/Harness [104, 101] extends MPI-1 semantics by incorporating MPI-2's and PVM's dynamic process model. To deal with failures, the pre-registered user MPI error handler is invoked on a leader node to coordinate the recovery. The error handler can then call the modified `MPI_Comm_Dup()` function to either rebuild the broken communicator or shrink the communicator to current size [2]. Legacy MPI-1-compliant programs need to be redesigned to use new features. This is because MPI-1 standard assumes static group membership and static process ranks, and it is possible that after recovery the process ranks change or become non-contiguous.

MPI-FT [106] deals with broken communicators by spawning redundant replacement processes and communicators at application launch, then switching to a good communicator when the current communicator is dead. MPI/FT [105] use redundancy in a slightly different way. Multiple copies of the same program are executed concurrently, and they perform the majority voting on every transmitted message. Processes also periodically vote on MPI's internal data structures to ensure the integrity of system state. Starfish [107] is an MPI implementation based on the Ensemble atomic group communication toolkit. The Ensemble keeps track of node health and group configuration. If any process crashes, it can optionally restart from the last checkpoint or deliver the node configuration change event to the Starfish. However, it is not clear how much modification must be done to the user program to utilize this dynamic configuration feature.

CoCheck [108] is a process migration environment based on Condor, a single process checkpointer Condor. It adopts the Chandy-Lamport's distributed snapshot algorithm (see §4.1) to achieve non-blocking checkpointing. Sankaran *et al* [111] incorporated Chandy-Lamport's scheme to LAM/MPI. Bronevetsky *et al* [112, 113] also adopted Chandy-Lamport, but implemented it at the user-level using the MPI function wrappers. This approach is MPI-library-independent, since their im-

[2] The original MPI-1 semantics for `MPI_Comm_Dup()` is simply to duplicate a communicator. It cannot add or delete nodes during run-time. In most MPI-1 implementations, the user MPI error handler is invoked only when erroneous parameters are passed to MPI calls.

plementation posits upon the host MPI library. MPICH-V [109, 110] provides fault tolerance by asynchronous checkpointing using Condor and message logging. It is different from previous approaches in that it has a set of special nodes called Channel Memory servers, and all messages must go through Channel Memory for logging before reaching their destination.

IBM's LoadLeveler [117] is a job management system for the IBM SP/2 mainframes. In addition to scheduling, LoadLeveler is capable of checkpointing and restarting parallel jobs either automatically or manually through user intervention. This functionality is implemented inside the IBM AIX operating system kernel. In automatic mode, the job description script specifies a checkpoint interval and storage locations to instruct LoadLeveler to take a snapshot of process images at specified intervals. In manual mode, the user program must call the Parallel Environment checkpointing APIs explicitly. However, LoadLeveler's checkpoint/restart functions have many restrictions. For example, the user program cannot use multi-threading, fork, interprocess communication, file locks, dynamically loaded libraries, device I/O, and any MPI library other than IBM's. Also, the saved job can restart only on machines with the same switch type and the same software environment.

System-level transparent checkpoint/restart capability can also be found on some shared memory machines such as SGI Origin series and Cray mainframes. The SGI IRIX operating system can checkpoint jobs through the `cpr` utility. A set of APIs is also provided for user-initiated checkpoint/restart [118]. As in IBM LoadLeveler's case, there are non-checkpointable objects such as network sockets connections and file pointers to special devices (e.g., tape drives). For an MPI job, `cpr` can checkpoint and restart it only if all of its processes reside on one host (each SGI mainframe can have up to 512 CPUs.) If an MPI job spans over several SGI mainframes, then network sockets must be used, making the job non-checkpointable.

The Cray mainframes (models T3E, SV1, and X1) have long supported system-level checkpoint/restart in their UNICOS operating system; both UNIX commands and APIs are provided [119]. By leveraging this checkpoint/restart capability, Cray systems also support advanced features like job migration and compaction[3], swapping/gang scheduling, and priority pre-emption.

[3]Compaction is the defragmentation of a job so all the CPUs it is using are continuous in topology sense. This can

Finally, Gropp and Lusk categorize ways to enhance the reliability of MPI [120] into four classes based on transparency to user applications:

1. The recovery is completely transparent to the user application.
2. The user application will be notified about the recovery and must perform user-level corrective actions such as loading the checkpoint.
3. The MPI library changes default MPI semantics or offers new MPI functions, such as the dynamic process management in the MPI-2 specification.
4. The user application aborts whenever failures occur and is restarted manually.

Each level has different cost of software development and performance overhead. Level 1 seems the most favorable but its implementation inevitably ties to certain system-level checkpoint library, which tends to save more data than necessary. At level 3, programmers have more liberty in dealing with failures. For example, in the master-slave computing model, the crash of a slave process does not hang the whole execution, so it can be dealt lightly. At this level considerable changes to the user application are required. Our fault-tolerant MPI implementation in Chapter 5 falls into level 2. The main reason is we observe that most scientific applications already have efficient user-level checkpoint/restart routines, which is exactly the level 2 requires. So with minimal programming efforts, we can make an MPI application resilient to failures.

10.4 Fault Tolerance Research for Large Systems

IBM recently began the Autonomic Computing initiative [121]. The name draws an analogy from the human body's autonomic nervous system, which frees the conscious brain from the burden of handling low-level but vital functions such as maintaining a constant temperature or regulating the heartbeat. The vision of autonomic *computing* is to reduce the ever-increasing management cost of large complex systems by enabling computers to manage themselves and handle varying workloads

greatly improve communication performance.

efficiently. Essentially it defines a dynamic resource allocation and performance optimization problem. From the fault tolerance perspective, systems should be able to recover quickly from crashes and freezes, or better yet, prevent them from occurring. Unrepairable failures can be viewed as resource loss, and the system should adapt to a smaller pool of resources by migrating workloads to a new facility where desired performance can be met.

One example prototype of autonomic computing is the IceCube database/storage server [122]. It is a physically three-dimensional, dense pile of Collective Intelligent Bricks. A "brick" is a PC with 12 hard drives in a very compact chassis. Two faces of a brick have 10 Gb/s network interface to adjacent bricks. The whole server can track the health, performance, and capacity of its bricks and work around any brick failure. Dead bricks are left in place until there is a maintenance time to replace.

Recovery-Oriented Computing [123] is an on-going effort to introduce fault tolerance into software that runs on potentially unreliable commodity hardware, with an emphasis on quick recovery of some Internet services. Their main argument is that fast recovery reduces the MTTR component in the "Availability = MTTF/(MTTF+MTTR)" formula, thereby increasing availability. One approach to ROC is to engineer an Internet service in a fine-grained modularized style such that software modules can be restarted individually without shutting down the whole service [124]. The premise behind this "recursive restartability" is that most nondeterministic software bugs or transient hardware faults can be fixed by reset. Another key tenet of ROC philosophy is the reduction of damage from operator errors by providing an "undo" function in software [125]. The "undo" idea has been applied to an e-mail server software to make it "invertible," allowing the roll-back of accidental message deletion, mailbox migration, and system misconfiguration.

Chapter 11

Conclusions and Future Work

We have designed and implemented a scalable, high-performance diskless checkpointing system for large computing systems. In this chapter, we summarize the contributions of this thesis and point out some future research directions.

11.1 Results

We have the analyzed failures of large computing systems to gain a solid understanding of failure behavior and its impact on applications running on them.

The failures are either soft (transient) errors or hard (permanent) errors. To analyze soft errors, we conducted a series of fault injection to simulate soft errors in registers, memory and communication networks. We found the applications are susceptible to soft errors, and in some case the error will lead the application to generate incorrect output.

For the hard errors, we analyzed the failure data from three large computing system. We computed descriptive reliability measures such as Mean Time To Repair, Mean Time Between Failure, availability. The result shows that software halts are the main source of outages, but hardware halts account for the most downtime. Further deep analysis of failure distribution and correlation show that in many cases the Time To Repair and the Time Between Failure exhibit heavy-tail distributions instead of exponential. Failures of different nodes could also co-occur in a short time frame due to

an environmental cause, e.g. power failure.

We have designed a scalable, high-performance diskless checkpointing system based on node partitioning. In diskless checkpointing, checkpoint is written to memory, which provides a speed advantage over disks. To guarantee data integrity, redundancy codes (parity codes or Reed-Solomon codes) are computed and stored on spares. The spare assumes the role of compute node in case of failure. I/O is made scalable by partitioning nodes and spares into small groups, and each group takes care of its own redundancy code calculation and node failure and recovery.

We have implemented diskless checkpointing. The implementation has two components: a lightweight in-memory file system with a UNIX-like I/O interface, and a codec module which generates redundancy codes. The in-memory file system resides in user process address space and is individualized to the user process it attaches to. The codec module is an independent program which uses MPI to assist transferring and merging encoding results across network.

Because recovery is usually initiated by the communication middleware layer, we have chosen an MPI library, LA-MPI, to add automatic restartability and scalable heartbeat monitoring capabilities. MPI-based scientific applications only need to add several lines of code to use the reliability features of LA-MPI.

We have verified diskless checkpointing using benchmarks and real scientific applications on large PC clusters. We explored various group sizes, spare count, and performance fine-tuning techniques in the experiment. The results show that diskless checkpointing has great scalability and can achieve 9-12 GB/s throughput with 448 clients in the single-spare-per-group setting and 4-6 GB/s in the two-spares-per-group setting. In our experiment, diskless checkpointing shows a clear performance advantage over disk-based parallel file systems when the number of clients increases to a certain point.

We have developed an analytical model to characterize performability of diskless checkpointing on future large systems. The model allows users to examine the reliability and performance overhead of different node partitioning and spare assignment schemes. As a technology projection, we have also evaluated the model numerically for fictitious systems with 5,000-20,000 nodes. We

found that more spares per group can increase reliability by a factor of 2.6 to 6.9. Individual node reliability also affects overall system reliability considerably.

11.2 Future Work

Lastly, we point some directions for future research work.

- We have stressed at the beginning of this thesis that diskless checkpointing does not eliminate disk-based file system. Instead, there could be an adaptive strategy which intelligently alternates among different storage choices and adjust checkpoint intervals based on real-time failure trend and system health analysis.

  Here is an example of how an adaptive checkpoint strategy can help improve reliability. It is known that a hard failure usually follows a period of performance degradation. Therefore, we could monitor the machine sensor readings such as CPU temperature and voltage fluctuation, chassis fan speed, packet retransmission rate, and disk I/O errors.[1]

  If there is any performance anomaly, we could take preventive actions by dynamically adjusting the checkpointing strategy. For example, if the CPU temperature exceeds normal operational range, we may either checkpoint more often or migrate processes to a stable node [127]. The latter can be done by emulating a hardware failure which will trigger recovery.

- The partition of nodes into groups is determined during the initialization and will not change ever since. The problem with this static scheme is that a catastrophic failure can easily occur if any group uses up all spares it has to cope failures. Ideally, a catastrophic failure should only occur when *all* spares are uses up. Therefore, to defer catastrophic failure and increase reliability, a dynamic scheme of reorganizing the groups can be used. The reorganization

[1] For example, SCSI drive re-reads and re-writes, and SMART readings. SMART (Self Monitoring and Reporting Technology) is a standardized specification for failure prediction adopted by the hard disk industry since 1995. It constantly monitors several key drive measurements related to impending failure and emits a warning if the values exceed thresholds [126].

should take place when a group has used up all its spares, and a re-partition of nodes and spares can follow.

- A closely related topic is failure recovery. The current ring-based heartbeat monitoring and recovery protocol in our enhanced LA-MPI library cannot handle two or more simultaneous failures of adjacent nodes in the ring topology. Developing an efficient, scalable failure detecting and recovery algorithm is of central concern in a fault-tolerant distributed systems.

- Diskless checkpointing itself can also be further optimized. We have mentioned the phase behavior found in many scientific applications. To exploit this phenomenon, the DLCKPT codec module could transfer the checkpoint data from memory to the disk-based file system during application's computation phase. Overlapping I/O with computation has been proven an effective way to hide I/O latency. Combining this technique with diskless checkpointing can guarantee better data integrity and recoverability.

- Reducing memory demand and checkpoint size is extremely important to the usability of diskless checkpointing. In this thesis, we have demonstrated that data compression could be somewhat helpful. Another promising approach is incremental checkpointing [128].

  In this approach, only the data that is modified since last checkpoint is saved. To identify modified data, two classes of techniques have been developed: page-based and hash-based. The former requires memory hardware and operating system support to manipulate the protection bits to identify dirty pages. For the latter scheme, memory is partition into blocks and a hash function runs over blocks to generate a set of compact "digest." By comparing the digest, one can find out changed blocks. We believe that by careful memory exclusion (no need to save the memory storage used by diskless checkpointing system), incremental checkpointing can be seamlessly integrated into diskless checkpointing or better yet, combined with data compression to further reduce checkpoint size.

References

[1] The BlueGene/L Team, "An overview of the BlueGene/L supercomputer," in *IEEE/ACM SuperComputing Conference (SC)*, 2002.

[2] C. Gerald and P. Wheatley, *Applied Numerical Analysis*. Addison Wesley, fifth ed., 1994.

[3] "MPI: A message-passing interface standard. Version 1.1." http://www.mpi-forum.org/docs/. Message Passing Interface Forum, 1995.

[4] A. Mirin *et al.*, "Very high resolution simulation of compressible turbulence on the IBM-SP system," in *IEEE/ACM SuperComputing Conference (SC)*, 1999.

[5] K. Koch, R. Baker, and R. Alcouffe, "Solution of the first-order form of the 3D discrete ordinates equation on a massively parallel processor," *Transactions of the American Nuclear Society*, vol. 65, 1992.

[6] A. Hoisie *et al.*, "A general predictive performance model for wavefront algorithms on clusters of SMPs," in *International Conference on Parallel Processing*, 2000.

[7] National Center for Supercomputing Applications, "Top ranking Linux supercomputer at NCSA delivers groundbreaking parallel I/O using the Lustre file system from Cluster File Systems, Inc.." Press Release, November 18, 2003.

[8] D. Patterson, G. Gibson, and R. H. Katz, "A case for Redundant Array of Inexpensive Disks (RAID)," in *ACM SIGMOD Conference of Management of Data*, 1988.

[9] T. Lin and D. Siewiorek, "Error log analysis: Statistical modeling and heuristic trend analysis," *IEEE Transactions on Reliability*, vol. 39, no. 4, 1990.

[10] C. Constantinescu, "Impact of deep submicron technology on dependability of VLSI circuits," in *International Conference on Dependable Systems and Networks (DSN)*, 2002.

[11] "DRAM soft error rate calculations," Tech. Rep. TN-04-28, Micron Technology, Inc, 1994.

[12] "Module mean time between failures (MTBF)," Tech. Rep. TN-04-45, Micron Technology, Inc, 1997.

[13] "Terrestrial cosmic rays and soft errors," *IBM Journal of Research and Development*, vol. 40, no. 1, 1996.

[14] Actel Corporation, "Understanding soft and firm errors in semiconductor devices," 2002.

[15] Tezzaron Semiconductor, “Soft errors in electronic memory - a white paper.” http://www.tachyonsemi.com/about/papers/.

[16] “Data integrity for Compaq NonStop Himalaya servers (white paper),” 1999.

[17] C. Constantinescu, “Teraflops supercomputer: Architecture and validation of the fault tolerance mechanisms,” *IEEE Transactions on Computers*, vol. 49, no. 9, 2000.

[18] T. Dell, “A white paper on the benefits of Chipkill-correct ECC for PC server main memory.” IBM Microelectronics Division, 1997.

[19] N. Boden *et al.*, “Myrinet: A gigabit per second local area network,” *IEEE Micro*, vol. 15, no. 1, 1995.

[20] W.-C. Feng, “The future of high-performance networking,” in *Workshop on New Visions for Large-Scale Networks: Research and Applications*, 2001.

[21] J. Stone and C. Partridge, “When the CRC and TCP checksum disagree,” in *ACM SIGCOMM*, 2000.

[22] A. Cataldo, “SRAM soft errors cause hard network problems.” EE Times, August 17, 2001.

[23] M. Hsueh, T. Tsai, and R. Iyer, “Fault injection techniques and tools,” *IEEE Computer*, vol. 30, no. 4, 1997.

[24] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “A high-performace, portable implementation of the MPI message passing interface standard,” *Parallel Computing*, vol. 22, no. 6, 1996.

[25] G. Allen *et al.*, “The Cactus Code: A problem solving environment for the Grid,” in *Proc. of High Performance Distributed Computing (HPDC)*, 2000.

[26] L. Kale *et al.*, “NAMD2: Greater scalability for parallel molecular dynamics,” *Journal of Computational Physics*, vol. 151, 1999.

[27] “http://www.csar.uiuc.edu.” The Center for Simulation of Advanced Rockets at the University of Illinois at Urbana-Champaign.

[28] P. Springer, “Analysis of application behavior during fault injection.” http://hpc.jpl.nasa.gov/PEP/pls/pubs.html.

[29] “IA-32 Intel architecture software developers manual, volume 1: Basic architecture.” Intel Corporation, 2003.

[30] N. Nethercote and J. Seward, “Valgrind: A program supervision framework,” *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, 2003.

[31] G. Burns, R. Daoud, and J. Vaigl, “LAM: An open cluster environment for MPI,” in *IEEE/ACM SuperComputing Conference (SC)*, 1994.

[32] R. L. Graham *et al.*, “A network-failure-tolerant message-passing system for terascale clusters,” Tech. Rep. LA-UR-02-892, Los Alamos National Laboratory, 2003.

[33] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," Tech. Rep. CMU-CS-96-181, Carnegie Mellon University, 1996.

[34] M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, 1985.

[35] J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, 1998.

[36] J. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software – Practice and Experience*, vol. 27, no. 9, 1997.

[37] J. Plank, "Note: Correction to the 1997 tutorial on Reed-Solomon coding," Tech. Rep. UT-CS-03-504, University of Tennessee, 2003.

[38] National Center for Supercomputing Applications, "Hierarchical data format." http://hdf.ncsa.uiuc.edu.

[39] World Wide Web Consortium, "Extensible markup language (xml) 1.0." W3C Recommendation, Feb. 10, 1998.

[40] D. Thain and M. Livny, "Parrot: Transparent user-level middleware for data-intensive computing," Tech. Rep. 1493(a), Computer Sciences Department, University of Wisconsin, 2003.

[41] K. Koch, "How does ASCI actually complete multi-month 1000-processor milestone simulations? (talk)," in *Conference on High Speed Computing*, 2002.

[42] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, 2005.

[43] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large compuing clusters," in *Conference on File and Storage Technologies (FAST)*, 2002.

[44] P. Schwan, "Lustre: Building a file system for 1,000 node clusters," in *The Linux Symposium*, (Ottawa, Canada), 2003.

[45] G. Gibson, B. Welch, D. Nagel, and B. Moxon, "Object storage: Scalable bandwidth for HPC clusters," in *Linux Clusters: the HPC Revolution Conference*, (San Jose, USA), 2003.

[46] E. Frachtenberg *et al.*, "STORM: Lightning-fast resource management," in *IEEE/ACM SuperComputing Conference (SC)*, 2003.

[47] "NPACI Rocks Cluster Distribution: Users guide," 2004.

[48] M. Jette and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Linux Clusters: the HPC Revolution Conference*, (San Jose, USA), 2003.

[49] E. Hendriks, "BProc: the Beowulf distributed process space," in *International Conference on Supercomputing (ICS)*, 2002.

[50] R. Tremaine *et al.*, "IBM Memory Expansion Technology (MXT)," *IBM Journal of Research and Development*, vol. 45, no. 2, 2001.

[51] A. Alameldeen and D. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," Tech. Rep. CS-TR-2004-1500, University of Wisconsin - Madison, 2004.

[52] J. Lee, W. Hong, and S. Kim, "Design and evaluation of a selective compressed memory system," in *International Conference On Computer Design (ICCD)*, 1999.

[53] M. Oberhumer, "LZO - a real-time data compression library." Available at http://www.oberhumer.com/opensource/lzo/.

[54] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *IEEE/ACM Supercomputing Conference (SC)*, 2003.

[55] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, no. 3, 2000.

[56] D. Zwillinger and S. Kokoska, *CRC Standard Probability and Statistics Tables and Formulae*. Chapman and Hall/CRC, 2000.

[57] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, 1974.

[58] E. Gelenbe, "On the optimum checkpoint interval," *Journal of the ACM*, vol. 26, no. 2, 1979.

[59] J. Daly, "A model for predicting the optimum checkpoint interval for restart dumps," in *Workshop on Terascale Performance Analysis, International Conference on Computational Science (ICCS)*, 2003.

[60] L. Wang *et al.*, "Modeling coordinated checkpointing for large-scale supercomputers," in *International Conference on Dependable Systems and Networks (DSN)*, 2005.

[61] G. E. Martin, *Counting: The Art of Enumerative Combinatorics*. Springer Verlag, 2001.

[62] A. Wood, "Predicting client/server availability," *IEEE Computer*, vol. 28, no. 4, 1995.

[63] N. Talagala and D. Patterson, "An analysis of error behaviour in a large storage system," Tech. Rep. USC/CSD-99-1042, University of California, Berkeley, 1999.

[64] J. L. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, third ed., 2002.

[65] J. Gray, "Why do computers stop and what can be done about it?," Tech. Rep. 85.7, Tandem Computers, 1985.

[66] J. Gray, "A census of Tandem system availability between 1985 and 1990," Tech. Rep. 90.1, Tandem Computers, 1990.

[67] J. Xu, Z. Kalbarczyk, and R. Iyer, "Networked Windows NT system field failure data analysis," in *IEEE Pacific Rim International Symposium on Dependable Computing*, 1999.

[68] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure data analysis of a LAN of Windows NT based computers," in *Symposium on Reliable Distributed Systems*, 1999.

[69] M. W. Mutka and M. Livny, "The available capacity of a privately owned workstation environmont," *Performance Evaluation*, vol. 12, no. 4, 1991.

[70] J. Brevik, D. Nurmi, and R. Wolski, "Quantifying machine availability in networked and desktop Grid systems," Tech. Rep. CS-2003-37, University of California at San Diego, 2003.

[71] D. Nurmi, J. Brevik, and R. Wolski, "Modeling machine availability in enterprise and wide-area distributed computing environments," Tech. Rep. CS-2003-28, University of California at San Diego, 2003.

[72] I. Lee and R. Iyer, "Faults, symptoms, and software fault tolerance in the Tandem GUARDIAN90 operating system," in *International Symposium on Fault-Tolerant Computing (FTCS)*, 1993.

[73] B. Levidow and B. Murphy, "Windows 2000 dependability," Tech. Rep. MSR-TR-2000-56, Microsoft Research, 2000.

[74] D. Tang, R. Iyer, and S. Subramani, "Analysis of the VAX/VMS error logs in multicomputer environments - a case study of software dependability," in *International Symposium on Software Reliability Engineering*, 1992.

[75] K. W. Lee, F. A. Tillman, and J. J. Higgins, "A literature survey of the human reliability component in a man-machine system," *IEEE Transactions on Reliability*, vol. 37, no. 1, 1988.

[76] A. Brown and D. Patterson, "To err is human," in *Workshop on Evaluating and Architecting System Dependability (EASY)*, 2001.

[77] R. Iyer and D. Rossetti, "Effect of system workload on operating system reliability: a study on IBM 3081," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, 1985.

[78] B. Murphy and T. Gent, "Measuring system and software reliability using an automated data collection process," *Quality and Reliability Engineering International*, vol. 11, no. 5, 1995.

[79] D. Tang and R. Iyer, "Failure analysis and modeling of a VAXcluster system," in *International Symposium on Fault-Tolerant Computing (FTCS)*, 1990.

[80] D. Oppenheimer and D. Patterson, "Architecture and dependability of large-scale Internet services," *IEEE Internet Computing*, vol. 6, no. 5, 2002.

[81] C. Labovitz, A. Ahuja, and F. Jahanian, "Experimental study of Internet stability and backbone failures," in *International Symposium on Fault-Tolerant Computing (FTCS)*, 1999.

[82] B. Chandra, M. Dahlin, L. Gao, and A. Nayate, "End-to-end WAN service availability," in *USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.

[83] D. R. Kuhn, "Sources of failure in the public switched telephone network," *IEEE Computer*, vol. 30, no. 4, 1997.

[84] L. Geppert, "World's most powerful supercomputer hits full stride," *IEEE Spectrum*, vol. 38, no. 11, 2001.

[85] M. Seager, "Operational machines: ASCI White (talk)," in *SOS 7th Workshop on Distributed Supercomputing*, 2003.

[86] J. Morrison, "The ASCI Q System at Los Alamos (talk)," in *SOS 7th Workshop on Distributed Supercomputing*, 2003.

[87] M. Levine, "NSF's terascale computing system (talk)," in *SOS 7th Workshop on Distributed Supercomputing*, 2003.

[88] W. T. Kramer, "How are we doing? A self-assessment of the quality of services and systems at NERSC (2001)." National Energy Research Scientific Computing Center, 2002.

[89] "NERSC system availability statistics." http://hpcf.nersc.gov/computers/stats/AvailStats/.

[90] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: the Google cluster architecture," *IEEE Micro*, vol. 23, no. 2, 2003.

[91] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, 1992.

[92] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-volatile memory for fast, reliable file systems," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[93] M. Wu and W. Zwaenepoel, "eNVy: A non-volatile, main memory storage system," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[94] P. M. Chen *et al.*, "The Rio File Cache: Surviving operating system crashes," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[95] A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning, "Conquest: Better performance through a disk/persistent-RAM hybrid file system," in *USENIX Annual Technical Conference*, 2002.

[96] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-memory based file system," in *USENIX Technical Conference*, 1995.

[97] A. Tanenbaum, *Modern Operating Systems*. Prentice-Hall, 2001.

[98] M. M. Silva, B. Veer, and J. G. Silva, "Checkpointing SPMD applications on Transputer networks," in *Scalable High-Performance Computing Conference (SHPCC)*, 1994.

[99] J. Plank and K. Li, "Faster checkpointing with N+1 parity," in *International Symposium on Fault-Tolerant Computing (FTCS)*, 1994.

[100] T. Chiueh and P. Deng, "Efficient checkpoint mechanisms for massively parallel machines," in *International Symposium on Fault-Tolerant Computing (FTCS)*, 1996.

[101] Z. Chen *et al.*, "Building fault survivable MPI programs with FT-MPI using diskless checkpointing," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[102] G. Zheng, L. Shi, and L. Kale, "FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *IEEE International Conference on Cluster Computing*, 2004.

[103] Z. Chen and J. Dongarra, "Numerically stable real-number codes based on random matrices," in *IEEE Information Theory Workshop*, 2004.

[104] G. E. Fagg *et al.*, "Fault tolerant communication library and applications for high performance computing," in *LACSI Symposium*, 2003.

[105] R. Batchu *et al.*, "MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing," in *International Symposium on Cluster Computing and the Grid (CCGrid)*, 2001.

[106] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou, "MPI-FT: portable fault tolerance scheme for MPI," *Parallel Processing Letters*, vol. 10, no. 4, 2000.

[107] A. Agbaria and R. Friedman, "Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations," in *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1999.

[108] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 1996.

[109] G. Bosilca *et al.*, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *IEEE/ACM SuperComputing Conference (SC)*, 2002.

[110] A. Bouteiller *et al.*, "MPICH-V2: A fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging," in *IEEE/ACM SuperComputing Conference (SC)*, 2003.

[111] S. Sankaran *et al.*, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," in *LACSI Symposium*, 2003.

[112] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Collective operations in an application-level fault tolerant MPI system," in *International Conference on Supercomputing (ICS)*, 2003.

[113] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of MPI programs," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[114] J. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *USENIX Winter Technical Conference*, 1995.

[115] M. Litzkow, M. Livny, and M. Mutka, "Condor – a hunter of idle workstations," in *International Conference on Distributed Computing Systems*, 1988.

[116] "http://www.nersc.gov/research/ftg/checkpoint/."

[117] IBM Corporation, "Workload management with LoadLeveler," 2001. Document Number SG24-6038-00.

[118] Silicon Graphics, Inc., "IRIX checkpoint and restart operation guide," 2003. Document Number 007-3236-009.

[119] Cray Inc., "Man page collection: UNICOS/mp library routines," 2003. Document Number S-2364-23.

[120] W. Gropp and E. Lusk, "Fault tolerance in MPI programs," *The International Journal of High Performance Computing Applications*, vol. 18, no. 3, 2004.

[121] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, 2003.

[122] IBM Corporation, "Autonomic storage." http://www.almaden.ibm.com/StorageSystems/autonomic_storage.

[123] D. Patterson *et al.*, "Recovery Oriented Computing (ROC): Motivation, definition, and case studies," Tech. Rep. USC/CSD-02-1175, University of California, Berkeley, 2002.

[124] G. Candea and A. Fox, "Recursive restartability: Turning the reboot sledgehammer into a scalpel," in *IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

[125] A. Brown and D. Patterson, "Undo for operators: Building an undoable e-mail store," in *USENIX Annual Technical Conference*, 2003.

[126] G. F. Hughes, J. F. Murray, K. Kreutz-Delgado, and C. Elkan, "Improved disk drive failure warnings," *IEEE Transactions on Reliability*, vol. 51, no. 3, 2002.

[127] P. Apparao and G. Averill, "Firmware-based platform reliability." Bird-of-Feather session, IEEE/ACM SC2004 High Performance Computing, Networking, and Storage Conference.

[128] S. Agarwal, R. Garg, M. Gupta, and J. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *International Conference on Supercomputing (ICS)*, 2003.

Author's Biography

Charng-da Lu received his Bachelor of Science and Master of Science in Computer Science from National Taiwan University in 1997 and the University of Illinois at Urbana-Champaign in 2002, respectively. His research interest includes system reliability, performance analysis, and mathematics. The following is his research work during 1999-2005.

- C. Lu and D. Reed, "Compact Application Signatures for Parallel and Distributed Scientific Codes" in *ACM/IEEE Conference on Supercomputing (SC)*, Baltimore, 2002.
- D. Reed, C. Mendes, and C. Lu, "Application Tuning and Adaptation," book chapter in *The Grid: Blueprint for a New Computing Infrastructure*, edited by I. Foster and C. Kesselman, 2nd edition, Morgan Kaufmann, November 2003.
- C. Lu and D. Reed, "Assessing Fault Sensitivity in MPI Applications" in *ACM/IEEE Conference on Supercomputing (SC)*, Pittsburgh, 2004. (Best Paper Award)
- D. Reed, C. Lu, and C. Mendes, "Reliability Challenges in Large Systems" in *Future Generation Computer Systems*, 2005. (to appear)
- C. Lu and D. Reed, "Failure Data Analysis of HPC Systems" (work in progress)