Assessing Fault Sensitivity in MPI Applications [*†]

Charng-da Lu
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Daniel A. Reed
Renaissance Computing Institute
University of North Carolina
Chapel Hill, North Carolina 27514

Abstract

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 and software are fully reliable becomes much less credible. Concomitantly, understanding application sensitivity to system failures is critical to establishing confidence in the outputs of large-scale applications.

Using software fault injection, we simulated single bit memory errors, register file upsets and MPI message payload corruption and measured the behavioral responses for a suite of MPI applications. These experiments showed that most applications are very sensitive to even single errors. Perhaps most worrisome, the errors were often undetected, yielding erroneous output with no user indicators. Encouragingly, even minimal internal application error checking and program assertions can detect some of the faults we injected.

1 Introduction

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 and software are fully reliable becomes much less credible. This is especially true for systems built from very low cost components that lack error correcting memory or end-to-end error detection and correction for message transport.

Hardware failures are usually classified as either hard errors or soft (transient) errors. Hard errors are permanent physical defects whose repair normally requires component replacement (e.g., a power supply or fan failure). Conversely, 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).

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).

Non-recoverable hardware failures are exacerbated by programming models with limited support for fault-tolerance. For scientific applications, MPI [1] is the most popular parallel programming model. However, the MPI standard does not specify mechanisms or interfaces for fault-tolerance - normally, all of an MPI application's tasks are terminated when any of the underlying nodes fails or becomes inaccessible.

In this paper, we examine the impact of soft errors on MPI applications by injecting faults into registers, the process address space, and MPI messages to simulate single-bit-flip errors. The remainder of this paper is organized as follows. In §2, we outline the rationale for fault assessment of commodity hardware and consider the common failure modes in memory and communication systems. This is followed by a description of our fault injection methodology in §3 and our experimental environment in §4. In §5, we describe the application suite and experimental results, followed by an overall assessment in §6 and §7. Finally, we discuss related work in §8 and conclude by summarizing results and future directions in §9.

[*]This work was supported in part 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.

2 COTS Failure Modes

The price/performance advantage of commercial off-the shelf (COTS) components has led many groups to assemble clusters containing thousands of nodes. Because the COTS market is very price sensitive, there is great pressure on manufacturers to eliminate any features not necessary for the intended, commodity market. 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.

As a motivating example, consider the Los Alamos ASCI Q system, with 33 TB of error correcting (ECC) memory. If one assumes one error every ten days for each 1 GB of memory and a 95 percent ECC coverage rate (see §2.1), the soft error rate is $33,000 \times 0.05$ or roughly 1,650 errors every ten days. If even some of these errors corrupt an application's data space or message payloads, then an application's outputs may be incorrect.

The Los Alamos Q system is constructed from high quality components, in contrast to those used to assemble many low cost, laboratory clusters. Hence, as low-cost COTS hardware becomes the standard building block, it is crucial that we understand the balance of component reliability and price relative to system reliability and application usability. In this light, we review the possible failure modes and probabilities for memory and message transmission, as a prelude to experimental analysis.

2.1 Memory Errors

In an analysis of system logs from workstation clusters, Lin and Siewiorek [2] 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 [3] to between 159-713 years for 16 and 64 Mb DRAMs [4, 5]. Overall, the HER has remained roughly constant as memory densities have increased [3].

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 [3].

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 [6]

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 [7] 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 [8] 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 [9].

A hardware-based fault injection experiment by Constantinescu [10] showed that 18 percent of errors are uncovered by ECC memory. 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 [8, 11]. 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.

2.2 Communication Errors

On parallel systems, transient errors can also occur when transmitting messages. Although the MPI 1.1

standard [1] 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 [12]) 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 [13]. In such situations, message data integrity is dependent on hardware-implemented, link-level checksums.

Stone and Patridge [14] 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 [15]. For long-running, communication-intensive codes on large systems, even a small link error rate can have serious implications for application reliability.

3 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 [16]. 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.

3.1 Software Environment

Our experimental target was Intel x86 systems running Linux 2.4, with the MPICH library [17] 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.

3.2 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. Modifying 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 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

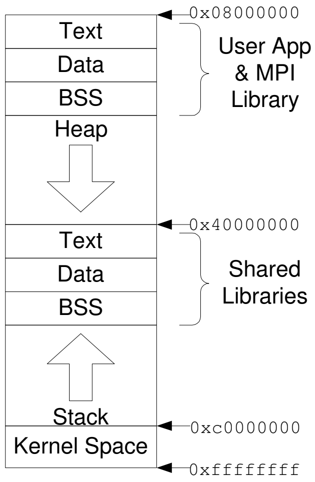


Figure 1: Linux Process Memory Model

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." By pointing a hook function at a user-defined function, we could change the behavior of the default `malloc`.

For example, our `malloc` wrapper invokes the GNU C library's `malloc` to allocate eight bytes more than the caller requested. These extra eight bytes, set by our `malloc` wrapper, store a 32-bit string (an identifier) and the size of the allocated memory chunk. The identifier indicates whether a memory chunk is allocated by the user application or the MPI library.

At entry to an MPI routine, a flag is set, and on exit, the flag is unset. Depending on the flag status, the `malloc` wrapper marks the allocated memory chunk as user or MPI. When the injector seeks to trigger a fault in the heap, starting at a random address, the injector looks for any memory chunk marked as user. Once located, a random bit in the chunk is flipped.

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.

3.3 Message Fault Injection

For MPI message injections, we modified the payload received immediately from the underlying communication software, as shown in Figure 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` 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 rebooted 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

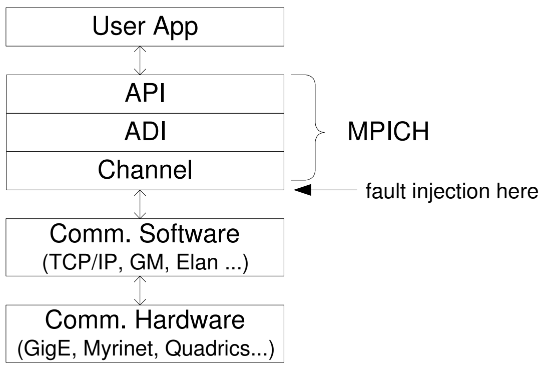


Figure 2: Fault Injection for MPI Messages

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.

4 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.

4.1 Test Applications

We used three scientific codes as test applications: Cactus Wavetoy [18], NAMD [19] and CAM [20]. Each of these codes is in use by computational scientists on a daily basis. To reduce the time needed to conduct experiments to tractable levels, we modified each application'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.

4.2 Application Profiles

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 1 shows the per-process application profiles.

For memory, we used the `objdump` and `nm` UNIX utilities to obtain the sizes of the text, data and BSS sections. We also used the `malloc` wrapper mentioned in §3.2 to obtain the heap size. For each of the three applications, the heap grows until it reaches a stable point, with limited variation about this fixed value. In Table 1, we reported this stable size. The stack size varied between 5-10 KB for all three applications.

For messages, we modified the MPICH library to measure and classify the incoming traffic at the Channel and ADI levels; see §3.3 and Figure 2. At these two levels, two kinds of messages are generated: control and data; both have 32-64 bytes of header. A control message only contains the header, because all control information is embedded within the header. A data message contains both header and payload. The payload carries the user data passed from MPI calls. As Table 1 shows, Wavetoy and NAMD exhibit one type of behavior, with the majority of the messages transmitting data. In contrast, CAM is dominated by control messages.

4.2.1 Cactus Wavetoy

Cactus [18] 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.

4.2.2 NAMD

NAMD [19] is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. NAMD is based on Charm++, an object-oriented parallel programming library [21]. In our tests, we used NAMD version 2.5b2 and configured Charm++ to use MPICH for message passing. Thus, Charm++ is considered a part of the user application, and it is subjected to fault injection.

	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 1: Per-Process Profiles of Test Applications

		Errors	**Error Manifestations (Percent)**		
Region	**Executions**	**(Percent)**	**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: Fault Injection Results (Cactus Wavetoy)

We used a 92,000 atom "apoa1" input problem, whose data size was 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.

4.2.3 CAM

The Community Atmosphere Model (CAM) [20] 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.

4.3 Fault Sampling

We used sampling theory to determine the number of injections used in the experiments. The fault injection space has three axes: the bit in the memory/register file or message payload, the particular MPI process and the injection time. If we denote each axis by b, m, and t, respectively, the size of this space is $b \times m \times t$.

The range of bit targets, b, ranges from 512 (sixteen 32-bit registers) to $150 \times 10^6 \times 8 \approx 10^9$ (total message volume) In turn, m, the MPI process identifier, ranged from 64-192 for our experimental environment. Each of our test applications executed for a few minutes, so t lies between 120 and 300.

In total, the injection space is at least $512 \times 64 \times 120 \approx 3.9 \times 10^6$. With constraints on the time one can devote to experiments, it is impossible to inject faults at each point in this space. As such, we used

		Errors	**Error Manifestations (Percent)**				
Region	**Executions**	**(Percent)**	**Crash**	**Hang**	**Incorrect**	**App Detected**	**MPI Detected**
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 3: Fault Injection Results (NAMD)

		Errors	**Error Manifestations (Percent)**				
Region	**Executions**	**(Percent)**	**Crash**	**Hang**	**Incorrect**	**App Detected**	**MPI Detected**
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 4: Fault Injection Results (CAM)

random sampling to choose a small number of points as our injection targets.

Sampling theory defines a *population* of N elements and a randomly drawn subset of the population called a *sample*, with size n. From a sample, the goal is to infer certain statistical properties of the population. In the simplest case, each element of the population belongs to either classes C_1 or C_2, Let P be the proportion of C_1 in the population and p be the proportion of C_1 in the sample. We want to estimate P from p.

The quality of the estimate is determined by the sample size n. It should be chosen such that the desired confidence interval $(1-\alpha)$ and the estimation error d are satisfied. Here, α is the risk of not obtaining such a confidence interval. Mathematically, we have

$$Pr(|P-p| < d) \geq 1-\alpha$$

That is, with at least $1-\alpha$ probability, p is not farther from P than d. When $N \gg n$ and assuming p is normally distributed, an approximation for n to achieve $1-\alpha$ accuracy is [22]:

$$n \geq P(1-P)(\frac{z_{\alpha/2}}{d})^2 \tag{1}$$

where $z_{\alpha/2}$ is the double-tailed α-point of a standard normal distribution. Note that the above formula is independent of population size N.

However, in (1) the right hand side depends on P, which is exactly the statistical property we want to estimate. This can be solved by oversampling [22], that is, taking $P = 0.5$ to maximize the right-hand side of (1). Therefore, our sample size becomes

$$n \geq 0.25(\frac{z_{\alpha/2}}{d})^2$$

If the population is divided into $k > 2$ disjoint classes C_1, C_2, …, C_k, the above equations still apply. If P_i and p_i are the proportion of C_i in the population and sample, respectively, then we can replace P and p in (1) by P_i and p_i and get the same result.

In our application of sampling theory, the population is the injection space and the classes represent different error manifestations; see §5.1. For each of the test applications, we performed 400-500 injections in most regions. With a confidence interval of 95 percent (i.e., $\alpha = 5\%$ and $z_{\alpha/2} = 1.96$) and using oversampling, the estimation error d is 4.4-4.9 percent.

5 Experimental Results

We executed each of the three applications (Cactus, NAMD and CAM) on our test clusters with the fault injection methodology described in §3. 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.

5.1 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.[1] 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.

[1]As mentioned in §2.2, MPI assumes the underlying network substrate is reliable, so the errors detectable by MPI concern the user application execution. See §6.2 for more details.

5.2 Results

Table 2 summarizes the results for Cactus Wavetoy. During our tests, no Application Detected or MPI Detected errors were encountered. Table 3 and 4 show the results for NAMD and CAM, respectively.

6 Failure Mode Analysis

Given the experimental data of §5, we turn to an analysis of the commonalities and differences across the three test applications. This is followed in §7, by an assessment of the implications for application design and system configuration.

6.1 Common Behaviors

6.1.1 Register Injections

Even a cursory examination of Tables 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 [23] 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. The main reason is not all floating-point registers are accessed, or the faults are overwritten before being accessed.

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 [24]. 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.

The x87 FPU instructions treat the FPU data registers as a register stack, which is addressed relative to the register atop the stack. To understand the effects of code generation on fault manifestation, we examined the assembly code generated by the compilers and found that the generated x87 FPU instructions generally use only four of the registers in the stack.

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

6.1.2 Memory Injections

The error rates for memory injections were consistently low, generally less than 10 percent, across the three applications. Table 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 [25] 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.[2] We recorded snapshots of text and data accesses periodically to understand temporal and spatial locality and their relation to error rates.

Tables 5–7 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 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.

[2]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.

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 5–7.

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 [23], 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.

6.2 Application Differences

In addition to similarities, there were also substantial differences in application behaviors and sensitivities to injected faults. First, both NAMD and CAM include internal consistency checks for NaN (Not a Number) for some key variables. Both codes reported many NaN errors as a consequences of our injecting faults into the floating-point registers.

When an error occurs, the tests within NAMD and CAM detect them with 47 percent and 26 percent probability, respectively; see Tables 3–4. After detecting NaN errors, both applications abort. In addition, both NAMD and CAM use sanity/bound checks and assertions on certain data structures to capture a fraction (3-7 percent and 4-13 percent, respectively) of faults that manifest in the Data, Text, Heap and Stack areas.

For example, in CAM, any moisture value below a minimum threshold can trigger a warning and abort the application. Although these tests still capture less than half of all injected errors, they highlight the critical importance of internal consistency checks when executing in environments where hardware errors may occur.

Both NAMD and CAM are quite sensitive to errors injected into message payloads, with 38 and 24 percent error rates, respectively. However, 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[3], 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.

[3]These checks are implemented inside NAMD, not Charm++.

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 [26] and LA-MPI [27]. 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.

In Cactus, only 3.1 percent of the perturbed message payloads have an adverse effect, which is much lower than NAMD can CAM. To understand this difference, we correlated Cactus's low perturbation with MPICH traffic structure, Cactus's message passing behavior and Cactus's output format.

Recall from §4.2 that 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 §4.2.1, 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.

7 Implications

From our experiments, one can draw several conclusions. First and most importantly, soft bit errors can dramatically and adversely affect application reliability on commodity parallel systems. 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 nondeterministic, making error identification difficult. The distinction between memory fault induced errors and small variations due to numerical roundoff are subtle and difficult to detect.

A considerable fraction of the induced errors lead to execution modes that do not terminate. Although determining if an execution will terminate is undecidable, simple progress metrics (e.g., FLOPS, messages per second or loop iterations per minute) can provide some practical detection mechanisms. If the application's performance drops below a user-defined threshold, it is very likely that the code is in a non-terminating mode.

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. As Table 1 shows, each NAMD process receives 13-33 MB of data during 2.8 minutes of execution. 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, a fraction of injected faults are captured by these checks. Use of internal checks is an important aspect of robust application implementation, but must be used wisely because excessive checks can still harm performance.

8 Related Work

There is a long and rich history of fault tolerance studies, ranging from hardware assessment though software measurement to algorithm-based fault tolerance. Below, we review representative work in related areas.

8.1 Fault Injection into Parallel Codes

In an early study of distributed memory parallel systems, Carreira *et al* [28] injected faults into the communication system of the T805 Transputer to study their impact on parallel applications. They used a kernel-mode fault injector and found that 5-30 percent of the injected faults caused the application to generate incorrect results. Mirroring our experience, bit errors in message packet headers almost always caused the applications to fail.

More recently, Constantinescu [10] used hardware-based fault injection to assess the reliability of the 9,000-processor ASCI Red system, which has ECC memory, parity, protocol checking, watchdog timers and message checksums to ensure data integrity. The injected faults were stuck-at-0/1's using a hardware probe at the IC pin level. During the fault experiments, the processors executed the Linpack benchmark, and the results were verified at the end of every iteration. Overall, Constantinescu found the error detection rate on the compute nodes was 80-84 percent, though error detection was dependent on the fault duration. Transients proved more difficult to detect, whereas longer faults led to application failures (hangs).

Our work differs in its use of a cost-effective user-mode software fault injector. We also used a suite of scientific codes and a large number of processes (64-192) to simulate a mid-sized cluster environment. We also used commodity Linux clusters and the MPI communication library, the *de facto* standard for parallel programming.

8.2 Software Solutions to Soft Errors

Many software-based reliability techniques have been developed to handle soft errors. To handle memory errors the text regions of application code, control-flow checking can monitor branches to determine if they deviate from a pre-generated control-flow signature [29]. Simultaneous multithreading (SMT) has also been exploited; two threads execute the same code, with one running slightly ahead of another. The trailing thread compares the values produced by both and triggers an error if there is mismatch [30].

In general, result checking and self-correction [31] seek efficient result checkers or correctors by exploiting program structure. Algorithm-based fault tolerance (ABFT) [32] techniques exploit the algorithmic structure of codes to create efficient, domain-specific detection schemes. Silva [33] reports that ABFT can detect almost all injected faults with only a ten percent performance penalty.

In some cases, one can exploit naturally fault tolerant algorithms [34] whose outputs are resilient to perturbation during the calculations. For example, iterative algorithms for solving systems of linear equations use successive approximations to obtain more accurate solutions at each step. A small error or lost data only slow convergence rather than leading to wrong results [35].

9 Conclusions

With increasing use of COTS components to construct large parallel systems, it is crucial that we understand the interplay of hardware component reliability and parallel application execution. Even a small chance of memory errors or communication errors can lead to application crashes, hangs or wrong outputs.

In this paper, we examined the impact of soft memory and message errors on MPI codes. We performed thousands of injections into registers, process address space, and MPI messages to simulate single-bit-flip errors. We found that registers and messages are particularly vulnerable to single-bit-flip faults, with an average 34.7 percent of fault manifestation rate. When a message fault manifests, the chance of producing an wrong output can be quite high, ranging from 28 to 71 percent for the three codes in our tests.

Application assertions and internal consistency checks can detect some of these errors, albeit at the expense of additional execution time. The MPI 1.1 library only supports very minimal error detection and recovery. Based on these results, we believe there should be a serious effort to redesign or enhance parallel applications and communication libraries with a renewed emphasis on fault tolerance, such that these applications can run successfully on large systems.

Acknowledgments

Special thanks go to Wei Deng for local cluster support and to NCSA for access to larger clusters.

References

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

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

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

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

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

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

[7] Actel Corporation, "Understanding soft and firm errors in semiconductor devices." http://www.actel.com/documents/SER_FAQ.pdf.

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

[9] "Data integrity for Compaq NonStop Himalaya servers (white paper)," 1999.

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

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

[12] N. J. Boden *et al.*, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, 1995.

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

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

[15] A. Cataldo, "SRAM soft errors cause hard network problems." EE Times, August 17, 2001.

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

[17] 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.

[18] G. Allen *et al.*, "The Cactus Code: A problem solving environment for the Grid," in *IEEE Symposium on High Performance Distributed Computing (HPDC)*, 2000.

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

[20] E. Kluzek *et al.*, "User's guide to NCAR CAM2.0," tech. rep., National Center for Atmospheric Research, Boulder, Colorado, 2002.

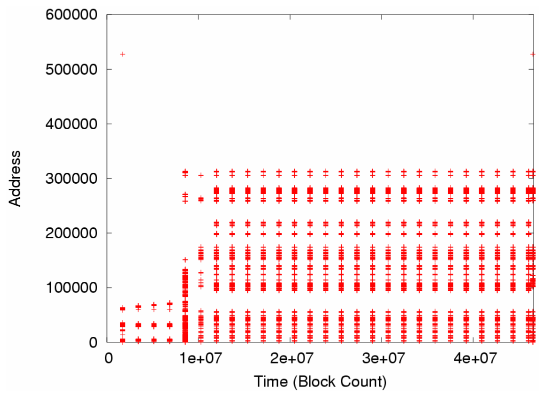
[21] L. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 1993.

[22] W. Cochran, *Sampling Techniques.* John Wiley & Sons, Inc, 1963.

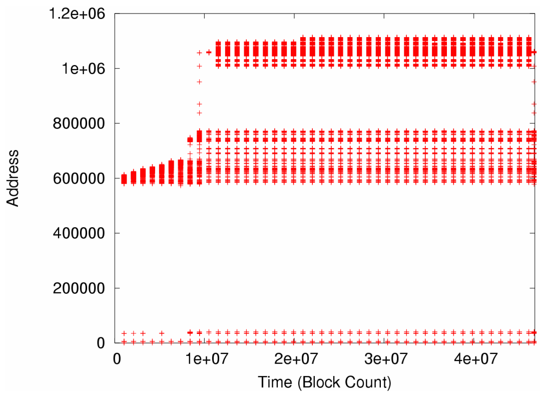
[23] P. Springer, "Analysis of application behavior during fault injection." http://hpc.jpl.nasa.gov/PEP/pls/pubs.html.

[24] "IA-32 Intel architecture software developers manual, volume 1: Basic architecture." Intel Corporation, 2003.

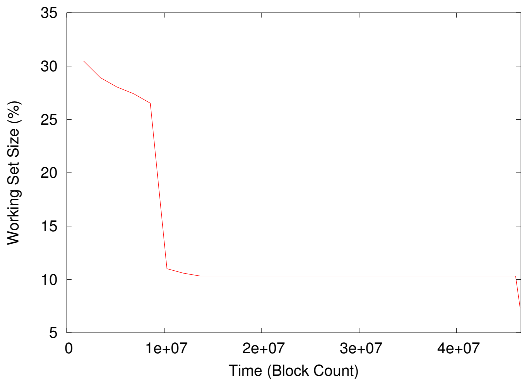
[25] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, 2003.



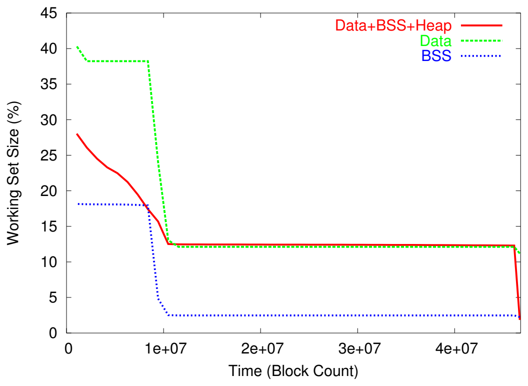
Text accesses



Data+BSS+Heap loads

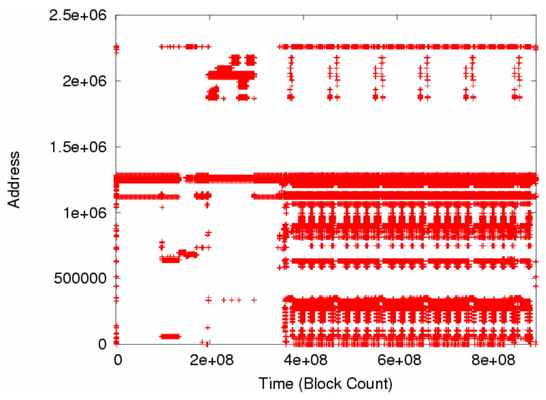


Text working set

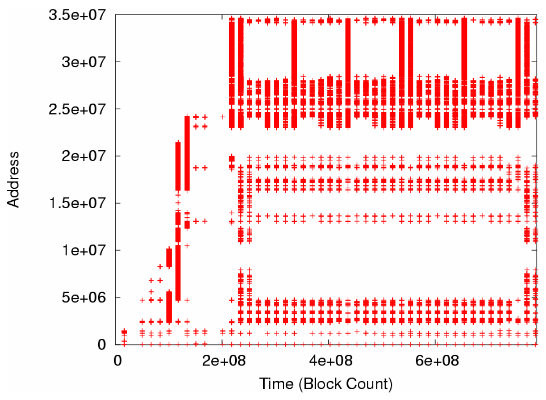


Data+BSS+Heap working set

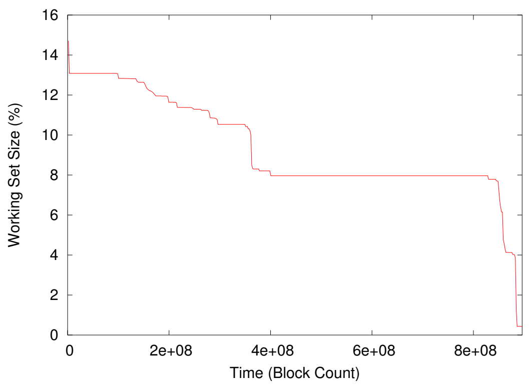
Table 5: Memory Trace of Cactus Wavetoy



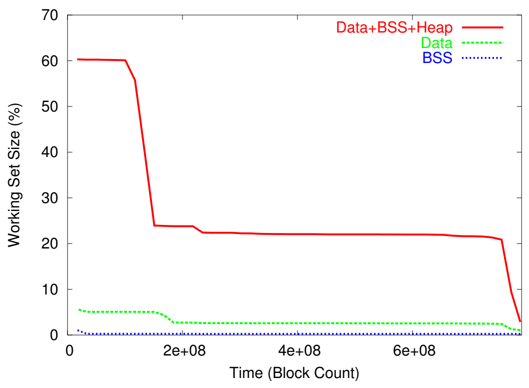
Text accesses



Data+BSS+Heap loads

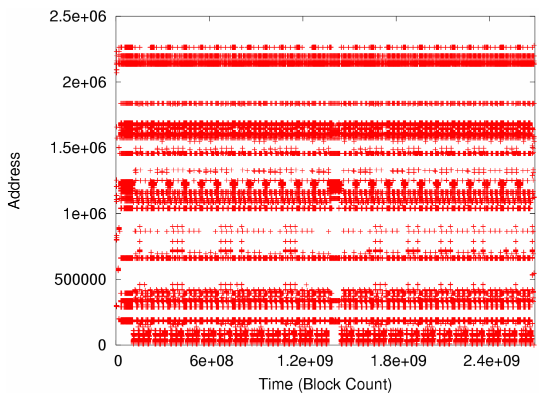


Text working set

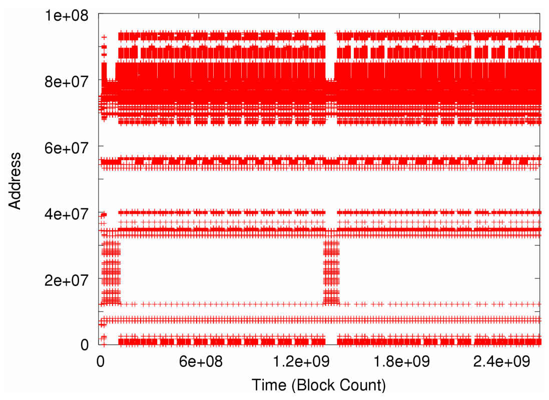


Data+BSS+Heap working set

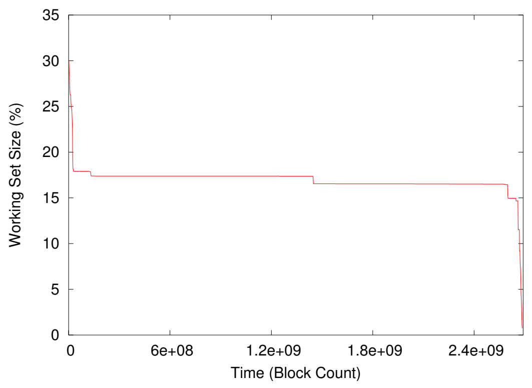
Table 6: Memory Trace of NAMD



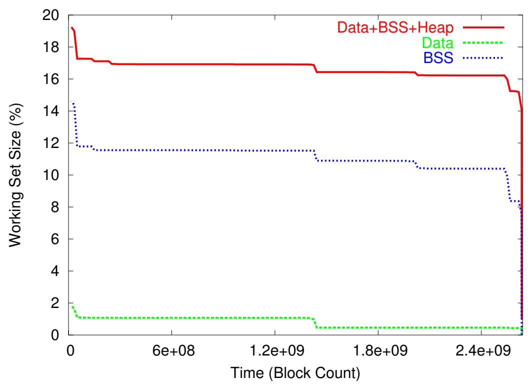
Text accesses



Data+BSS+Heap loads



Text working set



Data+BSS+Heap working set

Table 7: Memory Trace of CAM

[26] G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI," in *Supercomputing Conference (SC)*, 1994.

[27] R. Graham *et al.*, "A network-failure-tolerant message-passing system for terascale clusters," in *International Conference on Supercomputing (ICS)*, 2002.

[28] J. Carreira, H. Madeira, and J. Silva, "Assessing the effects of communication faults on parallel applications," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 1995.

[29] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control flow checking by software signatures," *IEEE Transactions on Reliability*, vol. 51, no. 2, 2002.

[30] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *International Symposium on Computer Architecture (ISCA)*, 2002.

[31] H. Wasserman and M. Blum, "Software reliability via run-time result-checking," *Journal of the ACM*, vol. 44, no. 6, 1997.

[32] K. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 33, no. 6, 1984.

[33] J. Silva, J. Carreira, H. Madeira, D. Costa, and F. Moreira, "Experimental assessment of parallel systems," in *Symposium on Fault-Tolerant Computing (FTCS)*, 1996.

[34] A. Geist and C. Engelmann, "Development of naturally fault tolerant algorithms for computing on 100,000 processors." http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf.

[35] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *Journal of the ACM*, vol. 25, no. 2, 1978.