A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility*

Xin Li Michael C. Huang Kai Shen
University of Rochester
{xinli@ece, mihuang@ece, kshen@cs}.rochester.edu

Lingkun Chu
Ask.com
lchu@ask.com

Abstract

Memory hardware reliability is an indispensable part of whole-system dependability. This paper presents the collection of realistic memory hardware error traces (including transient and non-transient errors) from production computer systems with more than 800 GB memory for around nine months. Detailed information on the error addresses allows us to identify patterns of single-bit, row, column, and whole-chip memory errors. Based on the collected traces, we explore the implications of different hardware ECC protection schemes so as to identify the most common error causes and approximate error rates exposed to the software level.

Further, we investigate the software system susceptibility to major error causes, with the goal of validating, questioning, and augmenting results of prior studies. In particular, we find that the earlier result that most memory hardware errors do not lead to incorrect software execution may not be valid, due to the unrealistic model of exclusive transient errors. Our study is based on an efficient memory error injection approach that applies hardware watchpoints on hotspot memory regions.

1 Introduction

Memory hardware errors are an important threat to computer system reliability [37] as VLSI technologies continue to scale [6]. Past case studies [27,38] suggested that these errors are significant contributing factors to whole-system failures. Managing memory hardware errors is an important component in developing an overall system dependability strategy. Recent software system studies have attempted to examine the impact of memory hardware errors on computer system reliability [11, 26] and security [14]. Software system countermeasures to these errors have also been investigated [31].

Despite its importance, our collective understanding about the rate, pattern, impact, and scaling trends of memory hardware errors is still somewhat fragmented and incomplete. The lack of knowledge on realistic errors has forced failure analysis researchers to use synthetic error models that have not been validated [11, 14, 24, 26, 31]. Without a good understanding, it is tempting for software developers in the field to attribute (often falsely) non-deterministic system failures or rare performance anomalies [36] to hardware errors. On the other hand, anecdotal evidence suggests that these errors are being encountered in the field. For example, we were able to follow a Rochester student's failure report and identify a memory hardware error on a medical System-on-Chip platform (Microchip PIC18F452). The faulty chip was used to monitor heart rate of neonates and it reported mysterious (and alarming) heart rate drops. Using an in-circuit debugger, we found the failure was caused by a memory bit (in the SRAM's 23rd byte) stuck at '1'.

Past studies on memory hardware errors heavily focused on transient (or soft) errors. While these errors received a thorough treatment in the literature [3,30,41,42, 44], non-transient errors (including permanent and intermittent errors) have seen less attention. The scarcity of non-transient error traces is partly due to the fact that collecting field data requires access to large-scale facilities and these errors do not lend themselves to accelerated tests as transient errors do [44]. The two studies of non-transient errors that we are aware of [10, 35] provide no result on specific error locations and patterns.

In an effort to acquire valuable error statistics in real-world environments, we have monitored memory hardware errors in three groups of computers—specifically, a rack-mounted Internet server farm with more than 200 machines, about 20 university desktops, and 70 PlanetLab machines. We have collected error tracking results on over 800 GB memory for around nine months. Our error traces are available on the web [34]. As far as we know, they are the first (and so far only) publicly available memory hardware error traces with detailed error addresses and patterns.

One important discovery from our error traces is that non-transient errors are at least as significant a source of reliability concern as transient errors. In theory, permanent hardware errors, whose symptoms persist over

*This work was supported in part by the National Science Foundation (NSF) grants CNS-0615045, CCF-0937571, CCF-0747324, and CNS-0719790. Kai Shen was also supported by an NSF CAREER Award CCF-0448413 and an IBM Faculty Award.

time, are easier to detect. Consequently they ought to present only a minimum threat to system reliability in an ideally-maintained environment. However, some non-transient errors are intermittent [10] (*i.e.*, whose symptoms are unstable at times) and they are not necessarily easy to detect. Further, the system maintenance is hardly perfect, particularly for hardware errors that do not trigger obvious system failures. Given our discovery of non-transient errors in real-world production systems, a holistic dependability strategy needs to take into account their presence and error characteristics.

We conduct trace-driven studies to understand hardware error manifestations and their impact on the software system. First, we extrapolate the collected traces into general statistical error manifestation patterns. We then perform Monte Carlo simulations to learn the error rate and particularly error causes under different memory protection mechanisms (*e.g.*, single-error-correcting ECC or stronger Chipkill ECC [12]). To achieve high confidence, we also study the sensitivity of our results to key parameters of our simulation model.

Further, we use a virtual machine-based error injection approach to study the error susceptibility of real software systems and applications. In particular, we discovered the previous conclusion that most memory hardware errors do not lead to incorrect software execution [11,26] is inappropriate for non-transient memory errors. We also validated the failure oblivious computing model [33] using our web server workload with injected non-transient errors.

2 Background

2.1 Terminology

In general, a fault is the cause of an error, and errors lead to service failures [23]. Precisely defining these terms ("fault", "error", and "failure"), however, can be "surprisingly difficult" [2], as it depends on the notion of the system and its boundaries. For instance, the consequence of reading from a defective memory cell (obtaining an erroneous result) can be considered as a *failure* of the memory subsystem, an *error* in the broader computer system, or it may not lead to any failure of the computer system at all if it is masked by subsequent processing. In our discussion, we use error to refer to the incidence of having incorrect memory content. The root cause of an error is the fault, which can be a particle impact, or defects in some part of the memory circuit. Note that an error does not manifest (*i.e.*, it is a *latent error*) until the corrupt location is accessed.

An error may involve more than a single bit. Specifically, we count all incorrect bits due to the same root cause as part of one error. This is different from the concept of a multi-bit error in the ECC context, in which case the multiple incorrect bits must fall into a single ECC word. To avoid confusions we call these errors word-wise multi-bit instead.

Transient memory errors are those that do not persist and are correctable by software overwrites or hardware scrubbing. They are usually caused by temporary environmental factors such as particle strikes from radioactive decay and cosmic ray-induced neutrons. *Non-transient* errors, on the other hand, are often caused (at least partially) by inherent manufacturing defect, insufficient burn-in, or device aging [6]. Once they manifest, they tend to cause more predictable errors as the deterioration is often irreversible. However, before transitioning into permanent errors, they may put the device into a marginal state causing apparently *intermittent* errors.

2.2 Memory ECC

Computer memories are often protected by some form of *parity-check code*. In a parity-check code, information symbols within a word are processed to generate *check* symbols. Together, they form the coded word. These codes are generally referred to as ECC (error correcting code). Commonly used ECC codes include SECDED and chipkill.

SECDED stands for *single-error correction, double-error detection*. Single error correction requires the code to have a Hamming distance of at least 3. In binary codes, it can be easily shown that r bits are needed for $2^r - 1$ information bits. For double-error detection, one more check bit is needed to increase the minimum distance to 4. The common practice is to use 8 check bits for 64 information bits forming a 72-bit ECC word as these widths are used in current DRAM standards (*e.g.*, DDR2).

Chipkill ECC is designed to tolerate word-wise multi-bit errors such as those caused when an entire memory device fails [12]. Physical constraints dictate that most memory modules have to use devices each providing 8 or 4 bits to fill the bus. This means that a chip-fail tolerant ECC code needs to correct 4 or 8 adjacent bits. While correcting multi-bit errors in a word is theoretically rather straightforward, in practice, given the DRAM bus standard, it is most convenient to limit the ECC word to 72 bits, and the 8-bit parity is insufficient to correct even a 4-bit symbol. To address this issue, one practice is to reduce the problem to that of single-bit correction by spreading the output of, say, 4 bits to 4 independent ECC words. The trade-off is that a DIMM now only provides 1/4 of the bits needed to fill the standard 64-data-bit DRAM bus, and thus a system needs a minimum of 4 DIMMs to function. Another approach is to use *b-adjacent* codes with much more involved matrices for parity generation and checking [7]. Even in this case, a typical implementation requires a minimum of 2 DIMMs. Due to these practical issues, chipkill ECC remains a technique used primarily in the server domain.

3 Realistic Memory Error Collection

Measurement results on memory hardware errors, particularly transient errors, are available in the literature. Ziegler *et al.* from IBM suggested that cosmic rays may cause transient memory bit flips [41] and did a series of measurements from 1978 to 1994 [30, 42, 44]. In a 1992 test for a vendor 4Mbit DRAM, they reported the rate of 5950 failures per billion device-hour. In 1996, Normand reported 4 errors out of 4 machines with a total of 8.8 Gbit memory during a 4-month test [29]. Published results on non-transient memory errors are few [10, 35] and they provide little detail on error addresses and patterns, which are essential for our analysis.

To enable our analysis on error manifestation and software susceptibility, we make efforts to collect realistic raw error rate and patterns on today's systems. Specifically, we perform long-term monitoring on large, non-biased sets of production computer systems. Due to the rareness of memory hardware errors, the error collection can require enormous efforts. The difficulty of acquiring large scale error data is aggravated by the efforts required for ensuring a robust and consistent collection/storage method on a vast number of machines. A general understanding of memory hardware errors is likely to require the collective and sustained effort from the research community as a whole. We are not attempting such an ambitious goal in this study. Instead, our emphasis is on the *realism* of our production system error collection. As such, we do not claim general applicability of our results.

Many large computer systems support various forms of error logging. Although it is tempting to exploit these error logs (as in some earlier study [35]), we are concerned with the statistical consistency of such data. In particular, the constantly improving efficacy of the error statistics collection can result in higher observed error rates over time by detecting more errors that had been left out before, while there could be no significant change in the real error rates. Also, a maintenance might temporarily suspend the monitoring, which will leave the faulty devices accumulate and later swarm in as a huge batch of bad chips once the monitoring comes back online. These factors all prevent a consistent and accurate error observation.

To ensure the statistical consistency of collected data, we perform proactive error monitoring under controlled, uniform collection methodology. Specifically, we monitor memory errors in three environments—a set of 212 production machines in a server farm at Ask.com [1], about 20 desktop computers at Rochester computer science department, and around 70 wide-area-distributed PlanetLab machines. Preliminary monitoring results (of shorter monitoring duration, focusing exclusively on transient errors, with little result analysis) were reported in another paper [25]. Here we provide an overview of our latest monitoring results on all error types. Due to factors such as machine configuration, our access privileges, and load, we obtained uneven amount of information from the three error monitoring environments. Most of our results were acquired from the large set of server farm machines, where we have access to the memory chipset's internal registers and can monitor the ECC-protected DRAM of all machines continuously. Below we focus our result reporting on the data obtained in this environment.

All 212 machines from the server farm use Intel E7520 chipset as memory controller hub [20]. Most machines have 4 GB DDR2 SDRAM. Intel E7520 memory controller is capable of both SECDED or Chipkill ECC. In addition to error detection and correction, the memory controller attempts to log some information about memory errors encountered. Unfortunately, this logging capability is somewhat limited—there are only two registers to track the addresses of two distinct errors. These registers will only capture the first two memory errors encountered. Any subsequent errors will not be logged until the registers are reset. Therefore, we periodically (once per hour) probe the memory controller to read out the information and reset the registers. This probing is realized through enhancements of the memory controller driver [5], which typically requires the administrative privilege on target machines.

Recall that when a memory cell's content is corrupted (creating a latent error), the error will not manifest to our monitoring system until the location is accessed. To help expose these latent errors, we enable hardware memory scrubbing—a background process that scans all memory addresses to detect and correct errors. The intention is to prevent errors from accumulating into more severe forms (*e.g.*, multi-bit) that are no longer correctable. It is typically performed at a low frequency (*e.g.*, 1.5 hours for every 1 GB) [20] to minimize the energy consumption and contention with running applications. Note that scrubbing does not help expose faults—writing varying values into memory does that. Since we monitored the machines for an extended period of time (9 months), the natural usage of the machines is likely to have exposed most (if not all) faults.

We collected error logs for a period of approximately 9 months (from November 30, 2006 to September 11, 2007). In the first 2 months we observed errors on 11 machines. No new errors were seen for 6 months and then 1 more erroneous machine appeared in the most recent

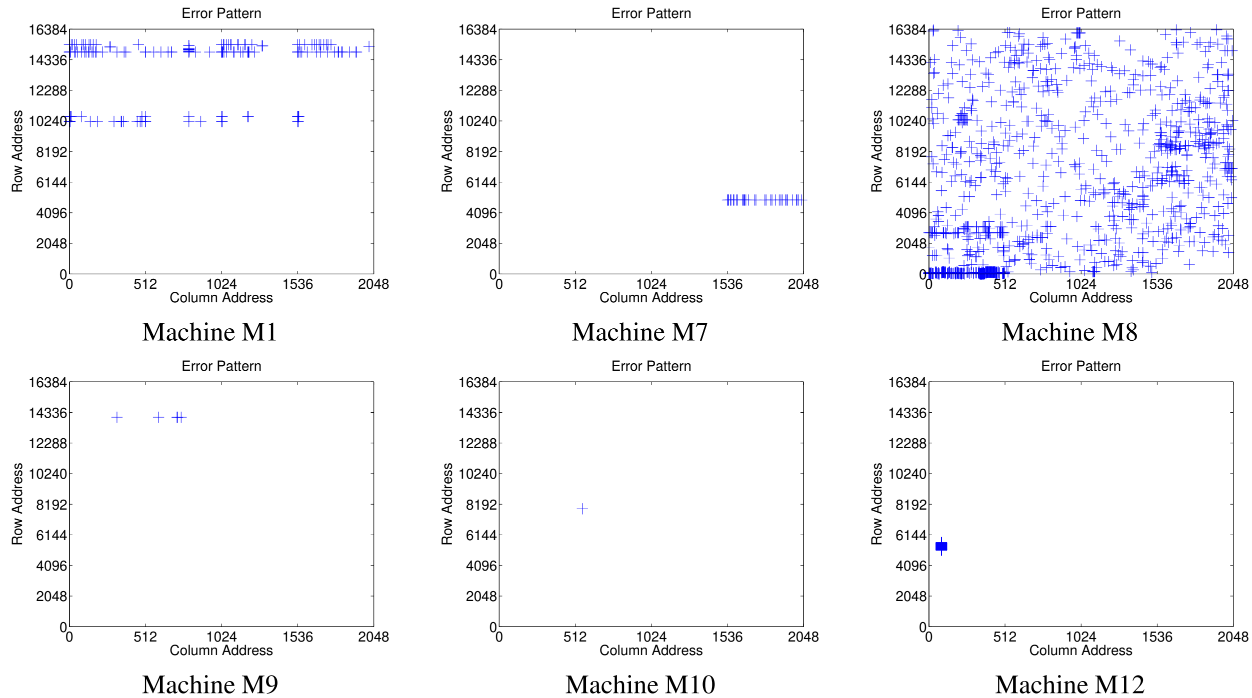

Figure 1. The visualization of example error patterns on physical memory devices. Each cross represents an erroneous cell at its row/column addresses. The system address to row/column address translation is obtained from the official Intel document [20].

month of monitoring. We choose 6 erroneous machines with distinct error patterns and show in Figure 1 how the errors are laid out on the physical memory arrays. Based on the observed patterns, all four memory error modes (single-cell, row, column, and whole-chip [4]) appear in our log. Specifically, M10 contains a single cell error. M7 and M12 represent a row error and a column error respectively. The error case on M1 is comprised of multiple row and columns. Finally, for machine M8, the errors are spread all over the chip which strongly suggests faults in the chip-wide circuitry rather than individual cells, rows, or columns. Based on the pattern of error addresses, we categorize all error instances into appropriate modes shown in Table 1.

While the error-correction logic can detect errors, it cannot tell whether an error is transient or not. We can, however, make the distinction by continued observation—repeated occurrences of error on the same address are virtually impossible to be external noise-induced transient errors as they should affect all elements with largely the same probability. We can also identify non-transient errors by recognizing known error modes related to inherent hardware defects: single-cell, row, column, and whole-chip [4]. For instance, memory row errors will manifest as a series of errors with addresses on the same row. Some addresses on this row may be caught on the log only once. Yet, the cause of that error is most likely non-transient if other cells on the same row indicate non-transient errors (logged multiple times). Consider M9 in Figure 1 as an example, there are five distinct error addresses recorded in our trace, two of which showed up only once while the rest were recorded multiple times. Since they happen on the same row, it is highly probable that they are all due to defects in places like the word line. We count them as a row error.

4 Error Manifestation Analysis

We analyze how device-level errors would be exposed to software. We are interested in the error manifestation rates and patterns (*e.g.*, multi-bits or single-bit) as well as leading causes for manifested errors. We explore results under different memory protection schemes. This is useful since Chipkill ECC represents a somewhat extreme trade-off between reliability and other factors (*e.g.*, performance and energy consumption) and may remain a limited-scope solution. In our memory chipset (Intel E7520) for example, to provide the necessary word length, the Chipkill design requires two memory channels to operate in a lock-stepping fashion, sacrificing throughput and power efficiency.

Machine	Cell	Row	Column	Row-Column	Whole-Chip
M1				1	
M2		1			
M3	1 (transient)				
M4	1				
M5	1 (transient)				
M6					1
M7		1			
M8					1
M9		1			
M10	1				
M11	1				
M12			1		
Total	5 (2 transient)	3	1	1	2

Table 1. Collected errors and their modes (single-cell, row, column, multiple rows and columns, or whole-chip errors). Two of the collected errors are suspected to be transient. Over a nine-month period, errors were observed on 12 machines out of the full set of 212 machines being monitored.

DRAM technology	DDR2
DIMM No. per machine	4
Device No. per DIMM	18
Device data width	x4
Row/Column/Bank No.	2^{14}/2^{11}/4
Device capacity	512 Mb
Capacity per machine	4 GB
ECC capability	None, SECDED, or Chipkill

Table 2. Memory configuration for our server farm machines.

4.1 Evaluation Methodology

We use a discrete-event simulator to conduct Monte-Carlo simulations to derive properties of manifested errors. We simulate 500 machines with the exact configuration as the Ask.com servers in Section 3. The detailed configuration is shown in Table 2. We first use the error properties extracted from our data to generate error instances in different memory locations in the simulated machines. Then we simulate different ECC algorithms to obtain a trace of manifested memory errors as the output. Our analysis here does not consider software susceptibility to manifested errors, which will be examined in Section 5. Below, we describe several important aspects of our simulation model, including temporal error distributions, device-level error patterns, and the repair maintenance model.

Temporal error distributions— We consider transient and non-transient errors separately in terms of temporal error distribution. Since transient errors are mostly induced by random external events, it is well established that their occurrences follow a memoryless exponential distribution. The cumulative distribution function of exponential distribution is $F(t) = 1 - e^{-\lambda t}$, which represents the probability that an error has already occurred by time t. The instantaneous error rate for exponential distribution is constant over time, and does not depend on how long the chip has been operating properly.

The non-transient error rate follows a "bathtub" curve with a high, but declining rate in the early "infant mortality" period, followed by a long and stable period with a low rate, before rising again when device wear-out starts to take over. Some study has also suggested that improved manufacturing techniques combined with faster upgrade of hardware have effectively made the wear-out region of the curve irrelevant [28]. In our analysis, we model 16 months of operation and ignore aging or wear-out. Under these assumptions, we use the oft-used Weibull distributions which has the following cumulative distribution function: $F(t) = 1 - e^{(t/\beta)^{\alpha}}$. The *shape parameter* α controls how steep the rate decreases, and the *scale parameter* β determines how "stretched out" the curve is. Without considering the wear-out region, the shape parameter in the Weibull distribution is no more than 1.0, at which point the distribution degenerates into an exponential distribution. The temporal error occurrence information in our data suggested a shape parameter of 0.11.

Device-level error patterns— For transient errors, prior studies and our own observation all point to the single-bit pattern. For non-transient errors, we have the 10 distinct patterns in our trace as templates. When a non-transient error is to be generated, we choose one out of these templates in a uniformly random fashion. There is a problem associated with using the exact template patterns—error instances generated from the same templates are always injected on the same memory location and thus they would always be aligned together to cause an uncorrectable error in the presence of ECC. To address this problem, we shift the error location by a random offset each time we inject an error instance.

Repair maintenance model— Our model requires a faulty device repair maintenance strategy. We employ an idealized "reactive" repair without preventive maintenance. We assume an error is detected as soon as it is exposed to the software level. If the error is diagnosed to be non-transient, the faulty memory module is replaced. Otherwise the machine will undergo a reboot. In our exploration, we have tried two other maintenance models that are more proactive. In the first case, hardware scrubbing is turned on so that transient errors are automatically corrected. In the second case, we further assume that the memory controller notifies the user upon detecting a correctable non-transient error so that faulty memory modules can be replaced as early as possible. We found these preventive measures have a negligible impact on our results. We will not consider these cases in this paper.

Below, Section 4.2 provides evaluation results using the above described model. Due to the small number of errors in the collected error trace, the derived rate and temporal manifestation pattern may not provide high statistical confidence. To achieve high confidence, we further study the sensitivity of our results to two model parameters—the Weibull distribution shape parameter for non-transient errors (Section 4.3) and the temporal error rate (Section 4.4).

4.2 Base Results

Here we present the simulation results on failures. The failure rates are computed as the average of the simulated operational duration. We describe our results under different memory protection schemes.

Figure 2(A) illustrates the failure rates and the breakdown of the causes when there is no ECC protection. In this case, any error will be directly exposed to software and cause a failure. As a result, we can study the errors in isolation. With our measurement, the transient error rate is 2006 FIT [1] for each machine's memory system. Depending on the operational time of the machines, the average non-transient error rates would vary, and so are the corresponding failure rates. Overall, for machines without ECC support, both transient and non-transient errors contribute to the overall error rate considerably.

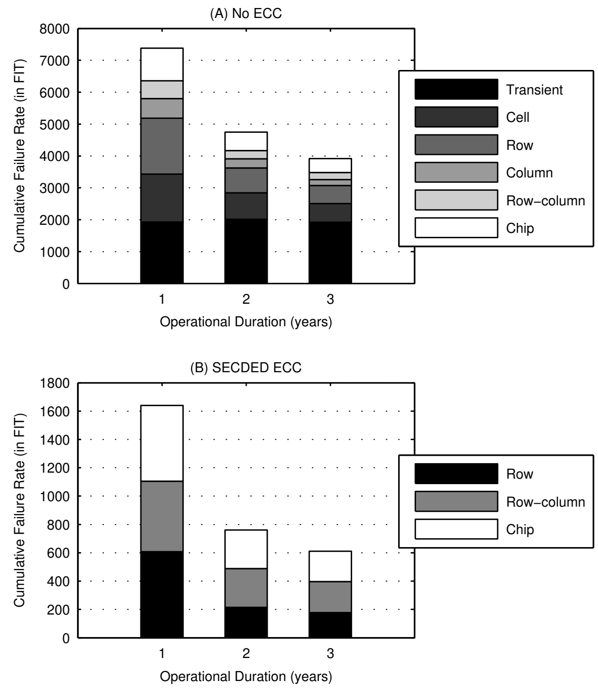

Figure 2. Base failure rates and breakdown causes for no ECC and SECDED ECC. Results (with varying machine operational durations) are for Section 4.2.

SECDED ECC can correct word-wise single-bit errors. For the errors in our trace, it could correct all but one whole-chip error, one row error, and one row-column error. These three cases all have multiple erroneous bits (due to the same root cause) in one ECC word, preventing ECC correction. Theoretically, a failure can also occur when multiple independent single-bit errors happen to affect the same ECC word (such as when a transient error occurs to an ECC word already having a single-bit non-transient error). However, since errors are rare in general, such combination errors are even less probable. In our simulations, no such instance has occurred. Figure 2(B) summarizes the simulation results.

When using the Chipkill ECC, as expected, the memory system becomes very resilient. We did not observe any uncorrected errors. This result echoes the conclusion of some past study [12].

[1] FIT is a commonly used unit to measure failure rates and 1 FIT equals one failure per billion device-hour. To put the numbers into perspectives, IBM's target FIT rates for servers are 114 for undetected (or silent) data corruption, 4500 for detected errors causing system termination, and 11400 for detected errors causing application termination [8]. Note that these rates are for the whole system including all components.

4.3 Shape Parameter Sensitivity for Non-Transient Error Distribution

To reach high confidence in our results, we consider a wide range of the Weibull shape parameters for the non-transient error temporal distribution and study the sensitivity of our results to this parameter. We use a machine operational duration of 16 months, which is the age of the Ask.com servers at the end of our data collection.

Prior failure mode studies in computer systems [16, 40], spacecraft electronics [17], electron tubes [22], and integrated circuits [19] pointed to a range of shape parameter values in 0.28–0.80. Given this and the fact that the Weibull distribution with shape parameter 1.0 degenerates to an exponential distribution, we consider the shape parameter range of 0.1–1.0 in this sensitivity study.

In both ECC mechanisms, the non-transient error rate depends on the Weibull shape parameter. The lower the shape parameter, the faster the error rate drops, and the lower the total error rate for the entire period observed. Note that the transient error rate also fluctuates a little because of the non-deterministic nature of our Monte-Carlo simulation. But the change of transient error rates does not correlate with the shape parameter. For no-ECC, as Figure 3(A) shows, for machines in their first 16 months of operation, the difference caused by the wide ranging shape parameter is rather insignificant.

In the case of SECDED shown in Figure 3(B), the impact of the Weibull shape parameter is a bit more pronounced than in the case of no ECC but is still relatively insignificant. Also, even though error rates are significantly reduced by SECDED, they are still within a factor of about five from those without ECC.

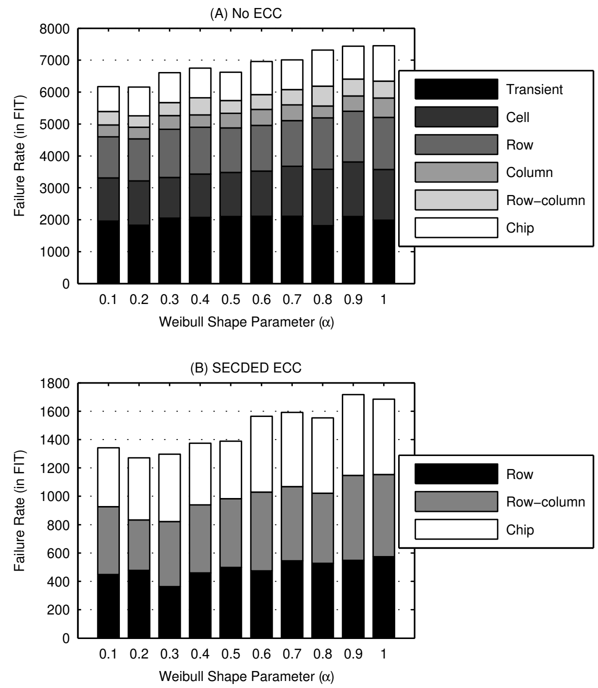

Figure 3. Failure rates and breakdown causes for no ECC and SECDED ECC, with varying Weibull shape parameter for non-transient error distribution. Results are for Section 4.3.

4.4 Statistical Error Rate Bounds

Due to the small number of device-level errors in our trace, the observed error rate may be quite different from the intrinsic error rate of our monitored system. To account for such inaccuracy, we use the concept of *p-value bounds* to provide a range of possible intrinsic error rates with statistical confidence.

For a given probability p, the p-value upper bound (λ_u) is defined as the intrinsic error rate under which $Pr\{X \leq n\} = p$. Here n is the actual number of errors observed in our experiment. X is the random variable for the number of errors occurring in an arbitrary experiment of the same time duration. And likewise, the p-value lower bound (λ_l) is the intrinsic error rate under which $Pr\{X \geq n\} = p$. A very small p indicates that given n observed errors, it is improbable for the actual intrinsic error rate λ to be greater than λ_u or less than λ_l.

Given p, the probability distribution of random variable X is required to calculate the p-value for our data. Thankfully, when the memory chips are considered identical, we can avoid this requirement. This is because in any time interval, their probability of having an error is the same, say q. Let N be the total number of memory chips operating, then the actual number of errors happening in this period, X, will be a random variable which conforms to binomial distribution: $P_{N,q}\{X = k\} = \binom{N}{k} q^k (1-q)^{N-k}$. When N is very large (we simulated thousands of chips), we can approximate by assuming N approaches infinity. In this case the binomial distribution will turn into Poisson distribution. For the ease of calculation, we shall use the form of Poisson distribution: $P_\lambda\{X = k\} = \frac{e^{-\lambda}\lambda^k}{k!}$, where $\lambda = q \cdot N$ is the expectation of X.

Based on the analysis above and the observed error rates, we have calculated the 1% upper and lower bounds. For instance, the transient error rate in non-ECC memory system is 2006 FIT as mentioned earlier. The corresponding 1%-upper-bound and 1%-lower-bound are 8429 FIT and 149 FIT respectively. The bounds on the various manifested error rates, derived from different raw error rates, are shown in Figure 4. From left to right, the bars show the 1%-lower-bound, the originally observed rate, and the 1%-upper-bound. As can be seen, for manifestations caused by non-transient errors, the two 1% bounds are roughly 2x to either direction of the observed rate. The ranges are narrow enough such that they have little impact to the qualitative conclusions.

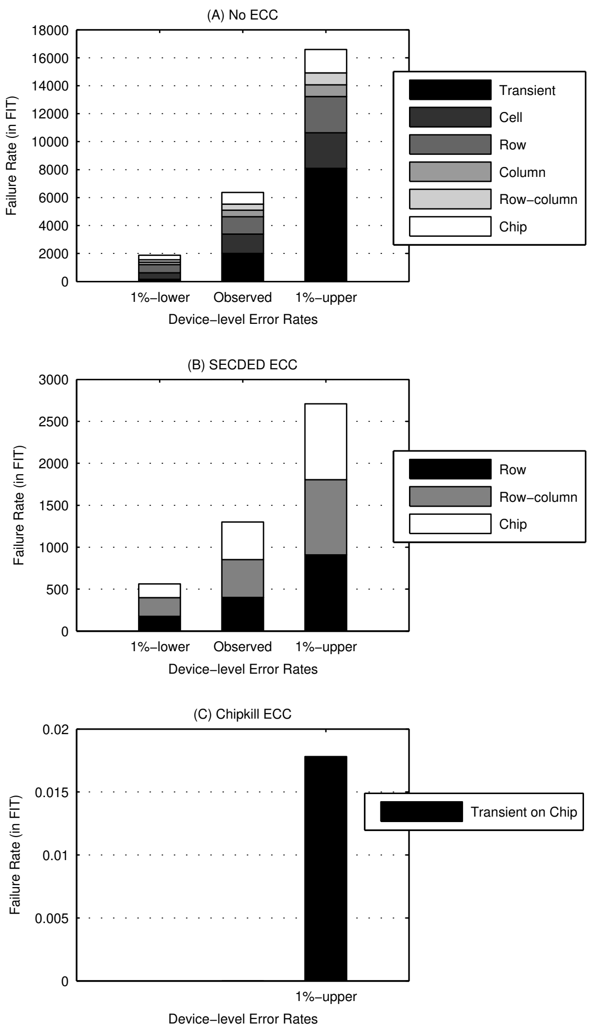

Figure 4. Manifested errors when input device-level error rates are the originally observed and 1%-lower/upper-bounds. Results are for Section 4.4.

For Chipkill ECC, the 1%-upper-bound offers a better chance to observe failures in the outcome of our simulation. With this increased rate, we finally produced a few failure instances (note there were none for Chipkill in the base simulations done in previous sub-sections). The patterns of the failures are shown in Figure 4(C). All of the failures here are caused by a transient error hitting an existing non-transient chip error.

4.5 Summary

We summarize our results of this part of the study:

- In terms of the absolute failure rate, with no ECC protection, error rates are at the level of thousands of FIT per machine. SECDED ECC lowers the rates to the neighborhood of 1000 FIT per machine. Chipkill ECC renders failure rates virtually negligible.

- Non-transient errors are significant (if not dominant) causes for all cases that we evaluated. Particularly on SECDED ECC machines, manifested failures tend to be caused by row errors, row-column errors, and whole-chip errors. With Chipkill ECC, the few failures occur when a transient error hits a memory device already inflicted with whole-chip non-transient errors.

- In terms of the error patterns, word-wise multi-bit failures are quite common.

Major implications of our results are that memory hardware errors exert non-negligible effects on the system dependability, even on machines equipped with SECDED ECC. Further, system dependability studies cannot assume a transient-error-only or single-bit-error-only model for memory hardware errors.

5 Software System Susceptibility

A memory error that escaped hardware ECC correction is exposed to the software level. However, its corrupted memory value may or may not be consumed by software programs. Even if it is consumed, the software system and applications may continue to behave correctly if such correctness does not depend on the consumed value. Now we shift our attention to the susceptibility of software systems and applications to memory errors. Specifically, we inject the realistic error patterns from our collected traces and observe the software behaviors. Guided by the conclusion of Section 4, we also take into account the shielding effect of ECC algorithms.

There is a rich body of prior research on software system reliability or security regarding memory hardware errors [11,14,24,26,31,33]. One key difference between these studies and ours is that all of our analysis and discussions build on the realism of our collected error trace. In this section, we tailor our software susceptibility evaluation in the context of recent relevant research with the hope of validating, questioning, or augmenting prior results.

5.1 Methodology of Empirical Evaluation

Memory Access Tracking and Manipulation To run real software systems on injected error patterns, we must accomplish the following goals. First, every read access to a faulty location must be supplied with an erroneous value following the injection pattern. This can be achieved by writing the erroneous value to each individual faulty address at the time of injection. Second, for every write access to a faulty location, if the error is non-transient, we must guarantee the erroneous value is restored right after the write. The injection is then followed by error manifestation bookkeeping. The bookkeeping facility has to be informed whenever a faulty address is accessed so that it would log some necessary information. The key challenge of such error injection and information logging is to effectively track and manipulate all the accesses to locations injected with errors (or *tracked locations*).

Traditional memory tracking approaches include:

- *Hardware watchpoints* [26]—employing hardware memory watchpoints on tracked locations. Due to the scarcity of hardware watchpoints on modern processors, this approach is not scalable (typically only able to track a few memory locations at a time).

- *Code instrumentation* [39]—modifying the binary code of target programs to intercept and check memory access instructions. This approach may incur excessive overhead since it normally must intercept all memory access instructions before knowing whether they hit on tracked memory locations. Further, it is challenging to apply this approach on whole-system monitoring including the operating system, libraries, and all software applications.

- *Page access control* [39]—applying virtual memory mechanism to trap accesses to all pages containing tracked memory locations and then manipulating them appropriately with accesses enabled. For this approach, it is important to reinstate the page access control after each page fault handling. This is typically achieved by single-stepping each trapped memory access, or by emulating the access within the page fault handler. This approach may also incur substantial overhead on *false memory traps* since all accesses to a page trigger traps even if a single location in the page needs to be tracked.

We propose a new approach to efficiently track a large number of memory locations. Our rationale is that although the whole system may contain many tracked locations exceeding the capacity of available hardware watchpoints, tracked locations within an individual page are typically few enough to fit. Further, the locality of executions suggests a high likelihood of many consecutive accesses to each page. By applying hardware watchpoints on tracked locations within the currently accessed *hot* page, we do not have to incur false traps on accesses to non-tracked locations within this page. At the same time, we enforce access control to other pages containing tracked locations. When an access to one such page is detected, we set the new page as hotspot and switch hardware watchpoint setup to tracked locations within the new page. We call our approach *hotspot watchpoints*. Its efficiency can be close to that of hardware watchpoints, without being subject to its scalability constraint. Note that it is possible that tracked locations within a page still exceed the hardware watchpoint capacity. If such a page is accessed, we fall back to the memory access single-stepping as in the page access control approach.

There is a chance for a single instruction to access multiple pages with tracked locations. For example, an instruction's code page and its data page may both contain tracked locations. If we only allow accesses to one tracked page at a time, then the instruction may trap on the multiple tracked pages alternately without making progress—or a livelock. We detect such livelock by keeping track of the last faulted program counter. Upon entering the page-fault handler, we suspect a livelock if the current faulting program counter address is the same as the last one. In such a situation we fall back to the memory access single-stepping while allowing accesses to multiple tracked pages that are necessary. It is possible that a recurring faulted program counter does not correspond to an actual livelock. In this case, our current approach would enforce an unnecessary instruction single-stepping. We believe such cases are rare, but if needed, we may avoid this slowdown by more precise tracking of execution progresses (*e.g.*, using hardware counters).

We should also mention a relevant memory monitoring technique called SafeMem [32], originally proposed for detecting memory leaks and corruptions. With modest modifications, it may be used for continual memory access tracking as well. SafeMem exploits the memory ECC mechanisms to trap accesses to all cachelines containing tracked locations. Because typical cacheline sizes (32–256 bytes) are smaller than the typical page size of 4 KB, false memory traps (those trapped memory accesses that do not actually hit tracked locations) under cacheline access control can be significantly fewer than that under page access control. Nevertheless, our hotspot watchpoints technique can further reduce the remaining false memory traps. In this case, hardware watchpoints will be set upon tracked locations within the current hot cacheline (or cachelines) instead of the hot page.

Error Monitoring Architecture If the error injection and monitoring mechanisms are built into the target system itself (as in [26]), these mechanisms may not behave reliably in the presence of injected memory errors. To avoid this potential problem, we utilize a virtual machine-based architecture in which the target system runs within a hosted virtual machine while the error injection and monitoring mechanisms are built in the underlying virtual machine monitor. We enable the shadow page table mode in the virtual machine memory management. Error injections only affect the shadow page tables while page tables within the target virtual machine are not affected. We also intercept further page table updates—we make sure whenever our faulty pages are mapped to any process, we will mark the protection bit in the corresponding page table.

In order to understand software system susceptibility to memory hardware errors, we log certain information every time an error is activated. Specifically, we record the access type (read or write), access mode (kernel or user), and the program counter value. For kernel mode accesses, we are able to locate specific operating system functions from the program counter values.

System Setup and Overhead Assessment Our experimental environment employs Xen 3.0.2 and runs the target system in a virtual machine with Linux 2.6.16 operating system. We examine three applications in our test: 1) the Apache web server running the static request portion of the SPECweb99 benchmark with around 2 GB web documents; 2) MCF from SPEC CPU2000—a memory-intensive vehicle scheduling program for mass transportation; and 3) compilation and linking of the Linux 2.6.23 kernel. The first is a typical server workload while the other two are representative workstation workloads (in which MCF is CPU-intensive while kernel build involves significant I/O).

We assess the overhead of our memory access tracking mechanism. We select error pattern M8 (illustrated in Figure 1), the one with most number of faulty bits in our overhead assessment. This error pattern consists of 1053 faulty 32-bit long words scattered in 779 pages, among which 668 pages contain only one erroneous word. Note that in this overhead evaluation, we only specify the spots to be tracked without actually flipping the memory bits. So the correctness of system and application executions should not be affected.

We compare the overhead of our approach to that of page access control. Results in Figure 5 suggest that our approach can significantly reduce the overhead compared to the alternative page access control approach. In particular, for Linux kernel build, our approach can reduce the execution time by almost a factor of four. The efficiency of our hotspot watchpoints approach also makes it a promising technique to support other utilization of memory access tracking [39] beyond the hardware error injection in this paper. Across the three applications, kernel build incurs the greatest amount of slowdown due to memory access tracking. We are able to attribute much (about two thirds) of the slowdown to the kernel function named `vma_merge` whose code section contains a tracked location. This function is triggered frequently by the GNU compiler when performing memory mapped I/O.

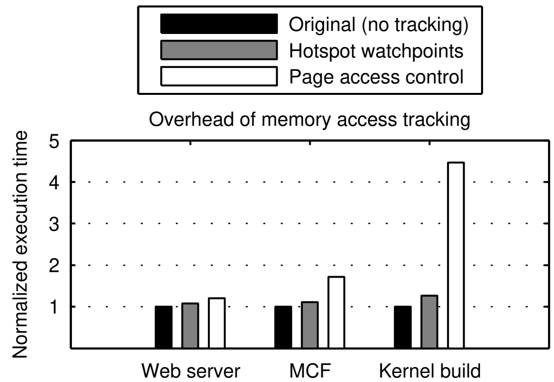

Figure 5. Benchmark execution time of our hotspot watchpoints approach, compared to the page access control approach [39]. The execution time is normalized to that of the original execution without memory access tracking. The slowdown was evaluated with the whole-chip error pattern of M8. Note for web server, the execution time is for requesting all the 2 GB data in a sweep-through fashion.

5.2 Evaluation and Discussion

Evaluation on Failure Severity Two previous studies [11, 26] investigated the susceptibility of software systems to transient memory errors. They reached similar conclusions that memory errors do not pose a significant threat to software systems. In particular, Messer *et al.* [26] discovered that of all the errors they injected, on average 20% were accessed, among which 74% are overwritten before being really consumed by the software. In other words, only 5% of the errors would cause abnormal software behaviors. However, these studies limited their scope for single-bit transient errors only. Our findings in Section 4 show non-transient errors are also a significant cause of memory failures. When these errors are taken into account, the previous conclusions may not stand intact. For example, non-transient errors may not be overwritten, and as a result, a portion of the 74% overwritten errors in [26] would have been consumed by the software system if they had been non-transient.

Table 3 summarizes the execution results of our three benchmark applications when non-transient errors are injected. Since our applications all finish in a short time (a few minutes), we consider these non-transient errors as permanent during the execution. In total we had 12 different error patterns. M3 and M5 are transient errors and therefore we do not include them in this result. M8 is so massive that as soon as it is injected, the OS crashes right away. We also exclude it from our results.

Application	Web server	MCF	Kernel build
No ECC			
M1 (row-col error)	WO	AC	AC
M2 (row error)	OK		
M4 (bit error)	OK		
M6 (chip error)	KC	WO	AC
M7 (row error)	WO	WO	
M9 (row error)	OK		
M10 (bit error)	OK		
M11 (bit error)			
M12 (col error)	WO		
SECDED ECC			
M1 (row-col error)	WO	WO	AC
M7 (row error)	WO	WO	

Table 3. Error manifestation for each of our three applications. The abbreviations in the table should be interpreted as follows, with descending manifestation severity: KC—kernel crash; AC—application crash; WO—wrong output; OK—program runs correctly. The blank cells indicate the error was not accessed at all.

Application	Web server	MCF	Kernel build
No ECC			
M1 (row-col error)	WO	AC	OK
M2 (row error)	OK		
M4 (bit error)	OK		
M6 (chip error)	KC	OK	OK
M7 (row error)	WO	OK	
M9 (row error)	OK		
M10 (bit error)	OK		
M11 (bit error)			
M12 (col error)	WO		
SECDED ECC			
M1 (row-col error)	WO	OK	OK
M7 (row error)	WO	OK	

Table 4. Error manifestation for each of our three applications, when the errors are made transient (thus correctable by overwrites). Compared to Table 3, many of the runs are less sensitive to transient errors and exhibit no mis-behavior at the application level.

The table includes results for both cases of no ECC and SECDED ECC. Since errors are extremely rare on Chipkill machines (see conclusions of Section 4), here we do not provide results for Chipkill. For no ECC, briefly speaking, out of the 27 runs, 13 have accessed memory errors and 8 did not finish with expected correct results. This translates to 48% of the errors are activated and 62% of the activated errors do lead to incorrect execution of software systems. In the SECDED case, single-bit errors would be corrected. Most errors (except M1 and M7) are completely shielded by the SECDED ECC. However, for the six runs with error patterns M1 and M7, five accessed the errors and subsequently caused abnormal behaviors.

Overall, compared to results in [26], non-transient errors evidently do cause more severe consequences to software executions. The reason for the difference is twofold— 1) non-transient errors are not correctable by overwriting and 2) unlike transient errors, non-transients sometimes involve a large number of erroneous bits. To demonstrate reason #1, we show in Table 4, when these errors are turned into transient ones (meaning they can be corrected by overwritten values), quite a few of the execution runs would finish unaffected.

Validation of Failure-Oblivious Computing This evaluation study attempts to validate the concept of failure-oblivious computing [33] with respect to memory hardware errors. The failure-oblivious model is based on the premise that in server workloads, error propagation distance is usually very small. When memory errors occur (mostly they were referring to out-of-bound memory accesses), a failure-oblivious computing model would discard the writes and supply the read with arbitrary values and try to proceed. In this way the error occurred will be confined within the local scope of a request and the server computation can be resumed without being greatly affected.

The failure-oblivious concept may also apply to memory hardware errors. It is important to know what the current operating system does in response to memory errors. Without ECC, the system is obviously unaware of any memory errors going on. Therefore it is truly failure-oblivious. With ECC, the system could detect some of the uncorrectable errors. At this point the system can choose to stop, or to continue execution (probably with some form of error logging). The specific choices are configurable and therefore machine dependent.

For our web server workload, we check the integrity of web request returns in the presence of memory errors. Table 5 lists the number of requests with wrong contents for the error cases. We only show error cases that trigger wrong output for the web server (as shown in Table 3). The worst case is M1, which caused 15 erroneous request returns (or files with incorrect content). However, this is still a small portion (about 0.1%) in the total 14400 files we have requested. Our result suggests that, in our tested web server workload, memory-hardware-error-induced failures tend not to propagate very far. This shows the promise of applying failure-oblivious computing in the management of memory hardware errors for server systems.

Discussion on Additional Cases Though error testing data from the industry are seldom published, modern

No ECC	
M1 (row-col error)	Wrong output (15 requests)
M7 (row error)	Wrong output (2 requests)
M12 (col error)	Wrong output (1 request)
SECDED ECC	
M1 (row-col error)	Wrong output (8 requests)
M7 (row error)	Wrong output (1 request)

Table 5. Number of requests affected by the errors in SPECweb99-driven Apache web server. We only show error cases that trigger wrong output for the web server (as shown in Table 3). We request 14400 files in the experiment.

commercial operating systems do advocate their countermeasures for faulty memory. Both IBM AIX [18] and Sun Solaris [38] have the ability to retire faulty memory when the ECC reports excessive correctable memory errors. Our results suggest that with ECC protection, the chances of errors aligning together to form an uncorrectable one is really low. However, this countermeasure could be effective against those errors that gradually develop into uncorrectable ones by themselves. Since our data does not have timestamps for most of the error instances, it is hard to verify how frequently these errors occur. On Chipkill machines [18], however, this countermeasure seems to be unnecessary since our data shows that without any replacement policy, Chipkill will maintain the memory failure rate at an extremely low level.

A previous security study [14] devised a clever attack that exploits memory errors to compromise the Java virtual machine (JVM). They fill the memory with pointers to an object of a particular class, and through an accidental bit flip, they hope one of the pointers can point to an object of another class. Obtaining a class A pointer actually pointing to a class B object is enough to compromise the whole JVM. In particular, they also provided an analysis of the effectiveness of exploiting multi-bit errors [14]. It appears that they can only exploit bit flips in a region within a pointer word (in their case, bit 2:27 for a 32-bit pointer). In order for an error to be exploitable, all the bits involved must be in the region. The probability that they can exploit the error decreases with the number of erroneous bits in the word. Considering that the multi-bit errors in our collected error trace are mostly consecutive rather than distributed randomly, we can be quite optimistic about successful attacks.

Another previous study [31] proposed a method to protect critical data against illegal memory writes as well as memory hardware errors. The basic idea is that software systems can create multiple copies of their critical data. If a memory error corrupts one copy, a consistency check can detect and even correct such errors. The efficacy of such an approach requires that only one copy of the critical data may be corrupted at a time. Using our collected realistic memory error patterns, we can explore how the placement of multiple critical data copies affects the chance for simultaneous corruption. In particular, about half of our non-transient errors exhibit regular column or row-wise array patterns. Therefore, when choosing locations for multiple critical data copies, it is best to have them reside in places with different hardware row and column addresses (especially row addresses).

6 Related Work

The literature on memory hardware errors can be traced back over several decades. In 1980, Elkind and Siewiorek reported various failure modes caused by low-level hardware fault mechanisms [13]. Due to the rareness of these errors, collecting error samples at a reasonable size would require a substantial amount of time and resource in field tests. Such field measurements have been conducted in the past (most notably by Ziegler at IBM) [29, 30, 43, 45]. These studies, however, have exclusively dedicated to transient errors and single-bit error patterns. Our previous study on transient error rates [25] also falls into this category.

Studies that cover non-transient errors are relatively few. In 2002, Constantinescu [10] reported error collection results on 193 machines. More recently, Schroeder *et al.* [35] examined memory errors on a larger number of servers from six different platforms. The large dataset enabled them to analyze statistical error correlations with environmental factors such as machine temperature and resource utilization. However, these studies provide no detail on error addresses or any criteria for categorizing transient and non-transient errors. Such results are essential for the error manifestation analysis and software susceptibility study in this paper.

Previous research has investigated error injection approaches at different levels. Kanawati *et al.* [21] altered target process images from a separate injection process that controls the target using `ptrace` calls. This is a user-level method that cannot inject errors to the operating system image. Li *et al.* [24] injected errors into hardware units using a whole-system simulator. This approach allows failure analysis over the whole system but the slow simulator speed severely limits the analysis scale.

Several studies utilized debugging registers for error injection at close-to-native speed. Gu *et al.* [15] focused on injecting faults in instruction streams (rather than memory error injection in our study). Carreira *et al.* [9] resorted to external ECC-like hardware to track the activation of memory errors whereas our approach is a software-only approach and therefore it can be applied on off-the-shelf hardware. In addition, they cannot monitor non-transient errors without completely single-stepping the execution. Messer *et al.* [26] also targeted transient errors. And their direct use of the watchpoint registers limited the number of simultaneously injected errors. In contrast, our hotspot watchpoint technique allows us to inject any number of transient and non-transient errors at high speed.

7 Conclusion

Memory hardware reliability is an indispensable part of whole-system dependability. Its importance is evidenced by a plethora of prior studies of memory error's impact on software systems. However, the absence of solid understanding of the error characteristics prevents software system researchers from making well reasoned assumptions, and it also hinders the careful evaluations over different choices of fault tolerance design.

In this paper, we have presented a set of memory hardware error data collected from production computer systems with more than 800 GB memory for around 9 months. We discover a significant number of non-transient errors (typically in the patterns of row or column errors). Driven by the collected error patterns and taking into account various ECC protection schemes, we conducted a Monte Carlo simulation to analyze how errors manifest at the interface between the memory subsystem and software applications. Our basic conclusion is that non-transient errors comprise a significant portion of the overall errors visible to software systems. In particular, with the conventional ECC protection scheme of SECDED, transient errors will be almost eliminated while only non-transient memory errors may affect software systems and applications.

We also investigated the susceptibility of software system and applications to realistic memory hardware error patterns. In particular, we find that the earlier results that most memory hardware errors do not lead to incorrect software execution [11, 26] may not be valid, due to the unrealistic model of exclusive transient errors. At the same time, we provide a validation for the failure-oblivious computing model [33] on a web server workload with injected memory hardware errors. Finally, as part of our software system susceptibility study, we proposed a novel memory access tracking technique that combines hardware watchpoints with coarse-grained memory protection to simultaneously monitor large number of memory locations with high efficiency.

Acknowledgments

We would like to thank Howard David at Intel for kindly interpreting the memory error syndromes discovered in our measurement. We also thank the anonymous USENIX reviewers and our shepherd Shan Lu for helpful comments on a preliminary version of this paper.

References

[1] Ask.com (formerly Ask Jeeves Search). http://www.ask.com.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[3] R. Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, 22(3):258–266, May 2005.

[4] M. Blaum, R. Goodman, and R. Mceliece. The reliability of single-error protected computer memories. *IEEE Trans. on Computers*, 37(1):114–118, 1988.

[5] EDAC project. http://bluesmoke.sourceforge.net.

[6] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov.–Dec. 2005.

[7] D. Bossen. b-adjacent error correction. *IBM Journal of Research and Development*, 14(4):402–408, 1970.

[8] D. Bossen. CMOS soft errors and server design. In *2002 Reliability Physics Tutorial Notes – Reliability Fundamentals*, pages 121.07.1–121.07.6, Dallas, Texas, Apr. 2002.

[9] J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Trans. Software Eng.*, 24(2):125–136, 1998.

[10] C. Constantinescu. Impact of deep submicron technology on dependability of VLSI circuits. In *Int'l Conf. on Dependable Systems and Networks*, pages 205–209, Bethesda, MD, June 2002.

[11] C. da Lu and D. A. Reed. Assessing fault sensitivity in MPI applications. In *Supercomputing*, Pittsburgh, PA, Nov. 2004.

[12] T. J. Dell. A white paper on the benefits of chipkill correct ECC for PC server main memory. *White paper*, 1997.

[13] S. A. Elkind and D. P. Siewiorek. Reliability and performance of error-correcting memory and register arrays. *IEEE Trans. on Computers*, (10):920–927, 1980.

[14] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symp. on Security and Privacy*, pages 154–165, Berkeley, CA, May 2003.

[15] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang. Characterization of linux kernel behavior under errors. In *Int'l Conf. on Dependable Systems and Networks*, pages 459–468, 2003.

[16] T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. In *ACM SIGMETRICS*, pages 217–227, Marina del Rey, CA, June 2002.

[17] H. Hecht and E. Fiorentino. Reliability assessment of spacecraft electronics. In *Annu. Reliability and Maintainability Symp.*, pages 341–346. IEEE, 1987.

[18] D. Henderson, B. Warner, and J. Mitchell. IBM POWER6 processor-based systems: Designed for availability. *White paper*, 2007.

[19] D. P. Holcomb and J. C. North. An infant mortality and long-term failure rate model for electronic equipment. *AT&T Technical Journal*, 64(1):15–31, January 1985.

[20] Intel E7520 chipset datasheet: Memory controller hub (MCH). http://www.intel.com/design/chipsets/E7520_E7320/documentation.htm.

[21] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Trans. Computers*, 44(2):248–260, 1995.

[22] J. H. K. Kao. A graphical estimation of mixed Weibull parameters in life-testing of electron tubes. *Technometrics*, 1(4):389–407, Nov. 1959.

[23] J. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *15th Int'l Symp. on Fault-Tolerant Computing*, pages 2–11, Ann Arbor, MI, June 1985.

[24] M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, Seattle, WA, Mar. 2008.

[25] X. Li, K. Shen, M. Huang, and L. Chu. A memory soft error measurement on production systems. In *USENIX Annual Technical Conf.*, pages 275–280, Santa Clara, CA, June 2007.

[26] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. J. F. Lie, D. Mannaru, A. Riska, and D. S. Milojicic. Susceptibility of commodity systems and software to memory soft errors. *IEEE Trans. on Computers*, 53(12):1557–1568, 2004.

[27] B. Murphy. Automating software failure reporting. *ACM Queue*, 2(8):42–48, Nov. 2004.

[28] F. R. Nash. *Estimating Device Reliability: Assessment of Credibility*. Springer, 1993. ISBN 079239304X.

[29] E. Normand. Single event upset at ground level. *IEEE Trans. on Nuclear Science*, 43(6):2742–2750, 1996.

[30] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM J. of Research and Development*, 40(1):41–50, 1996.

[31] K. Pattabiraman, V. Grover, and B. G. Zorn. Samurai: Protecting critical data in unsafe languages. In *Third EuroSys Conf.*, pages 219–232, Glasgow, Scotland, Apr. 2008.

[32] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *11th Int'l Symp. on High-Performance Computer Architecture*, pages 291–302, San Francisco, CA, Feb. 2005.

[33] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *6th USENIX Symp. on Operating Systems Design and Implementation*, pages 303–316, San Francisco, CA, Dec. 2004.

[34] Rochester memory hardware error research project. http://www.cs.rochester.edu/research/os/memerror/.

[35] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *ACM SIGMETRICS*, pages 193–204, Seattle, WA, June 2009.

[36] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS*, pages 85–96, Seattle, WA, June 2009.

[37] Sun Microsystems server memory failures. http://www.forbes.com/global/2000/1113/0323026a.html.

[38] D. Tang, P. Carruthers, Z. Totari, and M. W. Shapiro. Assessment of the effect of memory page retirement on system RAS against hardware faults. In *Int'l Conf. on Dependable Systems and Networks*, pages 365–370, Philadelphia, PA, June 2006.

[39] R. Wahbe. Efficient data breakpoints. In *5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 200–212, Boston, MA, Oct. 1992.

[40] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT system field failure data analysis. In *Pacific Rim Intl. Symp. on Dependable Computing*, pages 178–185, Hong Kong, China, Dec. 1999.

[41] J. Ziegler and W. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(16):776–788, Nov. 1979.

[42] J. Ziegler, M. Nelson, J. Shell, R. Peterson, C. Gelderloos, H. Muhlfeld, and C. Montrose. Cosmic ray soft error rates of 16-Mb DRAM memory chips. *IEEE Journal of Solid-State Circuits*, 33(2):246–252, Feb. 1998.

[43] J. F. Ziegler. Terrestrial cosmic rays. *IBM J. of Research and Development*, 40(1):19–39, 1996.

[44] J. F. Ziegler et al. IBM experiments in soft fails in computer electronics (1978–1994). *IBM J. of Research and Development*, 40(1):3–18, 1996.

[45] J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, T. J. O'Gorman, and J. M. Ross. Accelerated testing for cosmic soft-error rate. *IBM J. of Research and Development*, 40(1):51–72, 1996.