Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory

Yixin Luo   Sriram Govindan*   Bikash Sharma*   Mark Santaniello*   Justin Meza
Aman Kansal*   Jie Liu*   Badriddine Khessib*   Kushagra Vaid*   Onur Mutlu
Carnegie Mellon University, yixinluo@cs.cmu.edu, {meza, onur}@cmu.edu
*Microsoft Corporation, {srgovin, bsharma, marksan, kansal, jie.liu, bkhessib, kvaid}@microsoft.com

***Abstract*—Memory devices represent a key component of datacenter total cost of ownership (TCO), and techniques used to reduce errors that occur on these devices increase this cost. Existing approaches to providing reliability for memory devices pessimistically treat all data as equally vulnerable to memory errors. Our key insight is that there exists a diverse spectrum of tolerance to memory errors in new data-intensive applications, and that traditional one-size-fits-all memory reliability techniques are inefficient in terms of cost. For example, we found that while traditional error protection increases memory system cost by 12.5%, some applications can achieve 99.00% availability on a single server with a large number of memory errors *without* any error protection. This presents an opportunity to greatly reduce server hardware cost by provisioning the right amount of memory reliability for different applications.**

Toward this end, in this paper, we make three main contributions to enable highly-reliable servers at low datacenter cost. First, we develop a new methodology to quantify the tolerance of applications to memory errors. Second, using our methodology, we perform a case study of three new data-intensive workloads (an interactive web search application, an in-memory key–value store, and a graph mining framework) to identify new insights into the nature of application memory error vulnerability. Third, based on our insights, we propose several new hardware/software *heterogeneous-reliability* memory system designs to lower datacenter cost while achieving high reliability and discuss their trade-offs. We show that our new techniques can reduce server hardware cost by 4.7% while achieving 99.90% single server availability.

***Keywords*—memory errors, software reliability, memory architectures, soft errors, hard errors, datacenter cost, DRAM.**

I. Introduction

Warehouse-scale datacenters each consist of many thousands of machines running a diverse set of applications and comprise the foundation of the modern web [1]. While these datacenters are vital to the operation of companies such as Facebook, Google, Microsoft, and Yahoo!, reducing the cost of such large-scale deployments of machines poses a significant challenge to these and other companies. Recently, the need for reduced datacenter cost has driven companies to examine more energy-efficient server designs [2] and build their datacenter installations in cold environments to reduce cooling costs [3, 4] or use built-in power plants to reduce electricity supply costs [5].

There are two main components of the total cost of ownership (TCO) of a datacenter [1]: (1) capital costs (those associated with server hardware) and (2) operational costs (those associated with providing electricity and cooling). Recent studies have shown that capital costs can account for the majority (e.g., around 57% in [1]) of datacenter TCO, and thus represent the main impediment for reducing datacenter TCO. In addition, this component of datacenter TCO is only expected to increase going forward as companies adopt more efficient cooling and power supply techniques.

Of the dominant component of datacenter TCO (capital costs associated with server hardware), the cost of server processors and memory represents the key component—around 60% in modern servers [6]. Furthermore, the cost of the memory in today's servers is comparable to that of the processors, and is likely to exceed processor cost for data-intensive applications such as web search and social media services, which use in-memory caching to improve response time (e.g., a popular key–value store, Memcached, has been used at Google and Facebook [7] for this purpose).

Exacerbating the cost of memory in modern servers is the use of memory devices (such as dynamic random access memory, or DRAM) that provide error detection and correction. This cost arises from two components: (1) quality assurance testing performed by memory vendors to ensure devices sold to customers are of a high enough caliber and (2) additional memory capacity for error detection and correction. Device testing has been shown to account for an increasing fraction of the cost of memory for DRAM [8, 9]. The cost of additional memory capacity, on the other hand, depends on the technique used to provide error detection and correction.

Table 1 compares several common memory error detection and correction techniques in terms of which types of errors they are able to detect/correct and the additional amount of capacity/logic they require (which, for DRAM devices, whose design is fiercely cost-driven, is proportional to cost). Techniques range from the relatively low-cost (and widely employed) Parity, SEC-DED, Chipkill, and DEC-TED, which use different error correction codes (ECC) to detect and correct a small number of bits or chip errors, to the more expensive RAIM and Mirroring techniques that replicate some (or all) of memory to tolerate the failure of an entire DRAM dual in-line memory module (DIMM). The additional cost of memory with high error-tolerance can be significant (e.g., 12.5% with SEC-DED ECC and Chipkill and as high as 125% with Mirroring).

Yet even with well-tested and error-tolerant memory devices, recent studies from the field have observed a rising rate of memory error occurrences [13–15]. This trend presents an increasing challenge for ensuring high performance and high reliability in future systems, as memory errors can be detrimental to both.

In terms of performance, existing error detection and correction techniques incur a slowdown on each memory access due to their additional circuitry [15, 16] and up to

Technique	Error detection (correction)	Added capacity	Added logic
Parity	$2n$-1/64 bits (None)	1.56%	Low
SEC-DED	2/64 bits (1/64 bits)	12.5%	Low
DEC-TED	3/64 bits (2/64 bits)	23.4%	Low
Chipkill [10]	2/8 chips (1/8 chips)	12.5%	High
RAIM [11]	1/5 modules (1/5 modules)	40.6%	High
Mirroring [12]	2/8 chips (1/2 modules)	125%	Low

Table 1: Memory error detection and correction techniques. "X/Y Z" means a technique can detect/correct X out of every Y failures of Z.

an additional 10% slowdown due to techniques that operate DRAM at a slower speed to reduce the chances of random bit flips due to electrical interference in higher-density devices that pack more and more cells per square nanometer [17]. In addition, whenever an error is detected or corrected on modern hardware, the processor raises an interrupt that must be serviced by the system firmware (BIOS), incurring up to 100 µs latency—roughly 2000× a typical 50 ns memory access latency [18]—leading to unpredictable slowdowns.

In terms of reliability, memory errors can cause an application to slow down, crash, or produce incorrect results [19]. Software-level techniques such as the retirement of regions of memory with errors [15, 20–22] have been proposed to reduce the rate of memory error correction and prevent correctable errors from turning into uncorrectable errors over time. Hardware-level techniques, such as those listed in Table 1, are used to detect and correct errors without software intervention (but with additional hardware cost). All of these techniques are applied homogeneously to memory systems in a one-size-fits-all manner.

Our goal in this paper is to understand how tolerant different data-intensive applications are to memory errors and design a new memory system organization that matches hardware reliability to application tolerance in order to reduce system cost. Our key insight is that data-intensive applications exhibit a diverse spectrum of tolerance to memory errors—both within an application's data and among different applications—and that one-size-fits all solutions are inefficient in terms of cost. The main idea of our approach leverages this observation to classify applications based on their memory error tolerance and map applications to *heterogeneous-reliability* memory system designs managed cooperatively between hardware and software (for example, error-tolerant portions of data from an application may reside in inexpensive less-tested memory with no ECC with software-assisted data checkpointing, but error-vulnerable portions of its data should be placed in ECC memory) to reduce system cost. Toward this end, we provide the following ***contributions:***

- A new methodology to quantify the tolerance of applications to memory errors. Our approach measures the effect of memory errors on application correctness and quantifies an application's ability to mask or recover from memory errors.
- A comprehensive case study of the memory error tolerance of three data-intensive workloads: an interactive web search application, an in-memory key–value store, and a graph mining framework. We find that there exists an order of magnitude difference in memory error tolerance across these three applications.
- An exploration of the design space of new memory system organizations, which combine a heterogeneous mix of reliability techniques that leverage application error tolerance to reduce system cost. We show that our techniques can reduce server hardware cost by 4.7%, while achieving 99.90% single server availability.

II. Background and Related Work

A. Memory Errors and Mitigation Techniques

Modern devices use DRAM as their main memory. DRAM stores its data in cells in the form of charge in a capacitor. Over time, the charge in DRAM cells leaks and must be refreshed (every 64 ms in current devices). When data is accessed in DRAM, cell charge is sensed, amplified, and transmitted across a memory channel. In case any of the components used to store or transmit data fails, a memory error can occur.[1]

There are two main types of memory errors: (1) *soft* or *transient* errors and (2) *hard* or *recurring* errors.[2] Soft memory errors occur at random due to charged particle emissions from chip packaging or the atmosphere [28]. Hard memory errors may occur from physical device defects or wearout [13–15], and are influenced by environmental factors such as humidity, temperature, and utilization [13, 29, 30]. Hard errors typically affect multiple bits (for example, large memory regions and entire DRAM chips have been shown to fail [14, 15, 29]).

Various error correcting codes have been designed to mitigate these errors. ECC methods differ based on the amount of additional memory capacity required to detect and correct errors of different severity (Table 1 lists some of these techniques, discussed in Section I). The effects of hard errors can be mitigated in the operating system (OS) or BIOS by retiring affected memory regions when the number of errors in the region exceeds a certain threshold [15, 20–22]. Region retirement is typically done at the memory page granularity, ~4 KB. Retiring pages eliminates the performance overhead of the processor repeatedly performing detection and correction and also helps to prevent correctable errors from turning into uncorrectable errors.

A memory error will remain latent until the erroneous memory is accessed. The amount of time an error remains latent depends on how applications' data are mapped to physical memory and on the program's memory access pattern. For example, the frequency at which data is read determines how often errors are detected and corrected and the frequency at which data is written determines how often errors are masked by overwrites to the erroneous data.

B. Related Work

We categorize related research literature in memory error vulnerability and DRAM architecture into five broad classes: (1) characterizing application error tolerance, (2) hardware-based memory reliability techniques, (3) software-based memory reliability techniques, (4) exploiting application error tolerance, and (5) heterogeneous (hybrid) memory architectures. We next discuss these each in turn.

Classifying application error tolerance. Controlled error injection techniques based on hardware watchpoints [16, 31], binary instrumentation [32], and architectural simulation [33], have been used to investigate the impacts of memory errors on application behavior, including execution times, application/system crashes, and output correctness. These works have studied a range of applications including SPEC CPU benchmarks, web servers, databases, and scientific applications. In general, these works conclude that not all memory errors cause application/system crashes and can be tolerated with minimal difference in their outputs.

[1]For a detailed background on DRAM operation, we refer the reader to [23–25].

[2]Two recent studies [26, 27] examined the effects of *intermittent* and *access-pattern dependent* errors, which are increasingly common as DRAM technology scales down to smaller technology nodes.

We generalize this observation to data-intensive applications and leverage it to reduce datacenter TCO. Approximate computing techniques [34–36], where the precision of program output can be relaxed to achieve better performance or energy efficiency, offer further opportunities for leveraging the error-tolerance of application data, though these typically require changes to program source code.

Hardware-based memory reliability techniques. Various memory ECC techniques have been proposed (we list the most dominant ones in Table 1). Using eight bits, SEC-DED can correct a single bit flip and detect up to two bit flips out of every 64. DEC-TED is a generalization of SEC-DED that uses fourteen bits to correct two and detect three flipped bits out of every 64. Chipkill improves reliability by interleaving error detection and correction data among multiple DRAM chips [10]. RAIM [11] is able to tolerate entire DIMMs failing by storing detection and correction data across multiple DIMMs. Virtualized ECC [37] maps ECC to software-visible locations in memory so that software can decide what ECC protection to use. While Virtualized ECC can help reduce the DRAM hardware cost of memory reliability, it requires modification to the processor's memory management unit and cache(s).

Software-based memory reliability techniques. Previous works (e.g., [15, 22, 38]) have shown that the OS retiring memory pages after a certain number of errors can eliminate up to 96.8% of detected memory errors. These techniques, though they improve system reliability, still require costly ECC hardware for detecting and identifying memory pages with errors. Other works have attempted to reduce the impact of memory errors on system reliability by writing more reliable software [39], modifying the OS memory allocator [40], or using a compiler to generate a more error-tolerant version of the program [41, 42]. Other algorithmic solutions (e.g., memory bounds checks [43], watchdog timers [43], and checkpoint recovery [44–46]) have also been applied to achieve resilience to memory errors.

Exploiting application error tolerance. Flikker [47] proposed a technique to trade off DRAM reliability for energy savings. It relies on the programmer to separate application data into vulnerable or tolerant data. Less reliable mobile DIMMs have been proposed [48, 49] as a replacement for ECC DIMMs in servers to improve energy efficiency.

Heterogeneous (hybrid) memory architectures. Several works (e.g., [50–54]) explored the use of heterogeneous memory architectures, consisting of multiple different types of memories. These works were mainly concerned with either mitigating the overheads of emerging technologies or improving performance and power efficiency. They did not investigate the use of multiple devices with different error correction capabilities.

Compared to prior research, our work is the first to (1) perform a comprehensive analysis of memory error vulnerability *for data-intensive datacenter applications across a range of different memory error types* and (2) evaluate the cost-effectiveness of different *heterogeneous-reliability memory organizations* with hardware/software cooperation. Although our error injection-based vulnerability characterization methodology is similar to [16, 31], our analysis is done on a wider range of application parameters (such as memory write frequency and application access patterns). We also provide additional insights as to how errors are

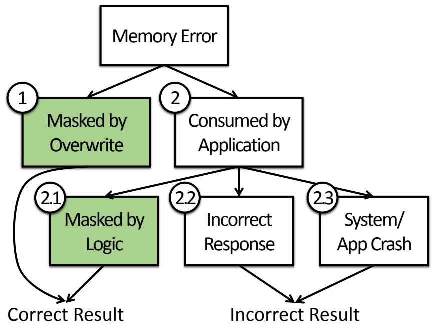


Figure 1: Memory error outcomes.

masked by applications. As such, our work studies memory error vulnerability for a new set of applications and proposes a new heterogeneous memory architecture to optimize datacenter cost.

III. Quantifying Application Memory Error Tolerance

We begin by discussing our methodology for quantifying the tolerance of applications to memory errors. Our methodology consists of three components: (1) characterizing the outcomes of memory errors on an application based on how they propagate through an application's code and data, (2) characterizing how safe or unsafe it is for memory errors to occur in different regions of an application's data, and (3) determining how amenable an application's data is to recovery in the event of an error. We next discuss each of these components in turn and describe their detailed implementation in Section IV.

A. Characterizing the Outcomes of a Memory Error

We characterize an application's vulnerability to a memory error based on its behavior after a memory error is introduced (we assume for the moment that no error detection or correction is being performed). Figure 1 shows a taxonomy of memory error outcomes. Our taxonomy is mutually exclusive (no two outcomes occur simultaneously) and exhaustive (it captures all possible outcomes). At a high level, a memory error may be either (1) masked by an overwrite, in which case it is never detected and causes no change in application behavior or (2) consumed by the application. In the case that an error is consumed by the application, it may either (2.1) be masked by application logic, in which case it is never detected and causes no change in application behavior, (2.2) cause the application to generate an incorrect response, or (2.3) cause the application or system to crash.

When we refer to the tolerance of an application to memory errors, we mean the likelihood that an error occurring in some data results in outcomes (1) or (2.1). Conversely, when we refer to the vulnerability of an application to memory errors, we mean the likelihood that an error occurring in some data results in outcomes (2.2) or (2.3).

B. Characterizing Safe Data Regions

An application's data is typically spread across multiple logical regions of memory to simplify OS-level memory management and protection. For example, a statically-allocated variable would be placed in a memory region called the *stack*, while a dynamically-allocated variable would be placed in a memory region called the *heap*. Table 2 lists several key memory regions and describes the types of

Region	The types of data that it stores
Heap	Memory for dynamically-allocated data.
Stack	Memory used to store function parameters and local variables.
Private	Pre-allocated private memory managed by the user (allocated by `VirtualAlloc` on Windows or `mmap` on Linux).
Other	Program code, managed heap, and so on, which are relatively small in size, and which we do not examine.

Table 2: Different application memory regions.

data that they store. How an application manipulates data in each of these regions ultimately determines its tolerance or vulnerability to memory errors, and so ideally we would like to quantify, for every address A in every region, how safe or unsafe it is (in terms of application vulnerability) for an error to occur at A.

Recall from Section II-A that the frequency at which data is read determines how often errors are detected (exposing an application to a potentially unsafe memory error) and the frequency at which data is written determines how often errors are masked by overwrites to the erroneous data (making the data safe for application consumption). For an address A, we define an address A's *unsafe duration* as the sum of time, across an application's execution time, between each *read* to A and the previous memory reference to A. Similarly, we define an address A's *safe duration* as the sum of time, across an application's execution time, between each *write* to A and the previous memory reference to A. Then, the $\textit{safe ratio} = \textit{safe duration}/(\textit{safe duration} + \textit{unsafe duration})$ for an address A is the fraction of time for which an error at A was never consumed by an application over its execution time.

A safe ratio closer to 1 implies that the address A is more frequently written than read, increasing the chances that an error at A will be masked; a safe ratio closer to 0 implies that the address A is more frequently read than written, increasing the chances that an error at A will be consumed by the application. We can generalize the concept of a safe ratio to regions of memory by computing the *average safe ratio* for all the addresses (or a representative sampling of the addresses) in a region.

C. Determining Data Recoverability

Even if an application is vulnerable to memory errors and its data is not particularly safe, it can still potentially recover from a memory error in its data by loading a clean copy of its data from persistent storage (i.e., flash or disk). This requires that a system is able to detect memory errors in hardware with Parity/ECC or in software by computing checksums over application data regions [19]. For example, once an error is detected, the OS can perform a recovery of clean data before repeating the memory access and returning control to the program. As such, *data recoverability* can improve application memory error tolerance. We have identified two strategies for data recoverability in applications: implicit recoverability and explicit recoverability.

Implicit recoverability is enabled by an application already maintaining a clean copy of its data in persistent storage. For example, the contents of a memory-mapped file can be re-read from disk. Though recovering data incurs the additional latency of reading from persistent storage, it only happens in the relatively uncommon case of a memory error.

Explicit recoverability is enabled by the system software automatically maintaining a clean copy of an application's memory pages in persistent storage. This can be realized by updating pages in persistent storage in tandem with updates to main memory. This may affect application performance and therefore may need to be applied only sparingly, to the most unsafe regions of memory.

We characterize data recoverability of a memory region by measuring the percentage of memory pages that are either implicitly or explicitly recoverable. A memory page is implicitly recoverable if the operating system can identify its mapped location on disk. A memory page is explicitly recoverable if, on average, it is written to less than once every five minutes.

Regardless of an application's recoverability strategy, for persistent hard errors (which cannot be corrected by recovering a clean copy of data), a technique such as page retirement can be used in conjunction with data recovery.

IV. Characterization Frameworks

We had three design goals when implementing our methodology for quantifying application memory error tolerance. First, due to the sporadic and inconsistent nature of memory errors in the field, we wanted to design a framework to emulate the occurrence of a memory error in an application's data in a controlled manner (to characterize its outcomes). Second, we wanted an efficient way to measure how an application accesses its data (to characterize safe and unsafe durations and determine data recoverability). Third, we wanted our framework to be easily adapted to other workloads or system configurations. To achieve all of these goals, we leveraged the support of existing software debuggers for modifying and monitoring arbitrary application memory locations. We next describe the details of the error emulation and access monitoring frameworks that we developed.

A. Memory Error Emulation Framework

Software debuggers (such as `WinDbg` [55] in Windows and `GDB` [56] in Linux) are typically used to examine and diagnose problems in application code. They run alongside a program and provide a variety of debugging features for software developers to analyze program behavior. For example, these debuggers can read and write the contents of an arbitrary memory location in an application. We leverage this ability to emulate both soft and hard errors.

```
1: addr = getMappedAddr()
2: data = *addr
3: bit = getRandBit()
4: data = data ∧ (1 << bit)
5: *addr = data
```

(a)

```
1: addr = getMappedAddr()
2: awatch(addr):
3:   print *addr
4:   print loadOrStore
5:   print time
```

(b)

Algorithm 1: Debugger pseudocode for (a) emulating single-bit soft memory errors and (b) measuring application memory access patterns.

Algorithm 1(a) lists the pseudocode for emulating a single-bit soft memory error in a random location in an application. In line 1, the `getMappedAddr()` function that we implement uses the debugger's ability to determine which memory addresses a program has data stored in to randomly select a valid byte-aligned application memory address, which it stores in the variable *addr*. In line 2, the value of the byte at *addr* is read and stored in the variable *data*. In line 3, the `getRandBit()` function that we implement uses a random number generator to select a bit index that determines which bit in a byte will contain an error, and stores this index in the variable *bit*. Line 4 performs the actual injection of the error by flipping the bit at index *bit* in *data* (using a left shift and an XOR operation). Finally, line 5 stores the modified version of *data* back to the location *addr*.

We generalize Algorithm 1(a) to generate multi-bit soft errors by repeatedly performing lines 3–4 with different values of *bit*. To emulate single- and multi-bit hard errors we perform the same approach, but to ensure that an application does not overwrite the emulated hard error, every 30 ms we check if the contents of *data* has changed, and if it has, we re-apply the hard error. We chose this approach so as to keep our emulation infrastructure reasonably fast while still emulating the features of hard errors.

Note that Algorithm 1(a) always flips the bit selected by *bit* in *data*. While it is true that some failure modes may only cause the value of a DRAM cell to change in one direction (only from 0 → 1 or only from 1 → 0 [28]), modern DRAM devices have been shown to contain regions of cells that can be induced to flip in either direction [27, 57], and so we do not impose any constraints on the direction of bit flips.

One difference between the errors that we emulate in software and memory errors in hardware is that our technique *immediately* makes memory errors visible throughout the system. In contrast, memory errors in hardware may take a while to become visible to system as processor caches may prevent the erroneous data in memory from being accessed by serving the correct data from the cache data store. Due to this, our methodology provides a more conservative estimate of application memory error tolerance.

Tying everything together, Figure 2 shows a flow diagram illustrating the five steps involved in our error emulation framework. We assume that the application under examination has already been run outside of the framework and its expected output has been recorded. The framework proceeds as follows. (1) We start the application under the error injection framework. (2) The debugger injects the desired number and type of errors. (3) We initiate the connection of a client and start performing the desired workload. (4) Throughout the course of the application's execution, we check to see if the machine has crashed (we deem an application in a crashed state if it fails to respond to ≥ 50% of the client's requests), corresponding to outcome (2.3) in Figure 1; if it has, we log this outcome and proceed to step (1) to begin testing once again. (5) If the application finishes its workload, we check to see if its output matches the expected results, corresponding to either outcome (1) or (2.1) in Figure 1.[3] If its output does not match the expected results, corresponding to outcome (2.2) in Figure 1, we log this outcome and proceed to step (1) to test again.

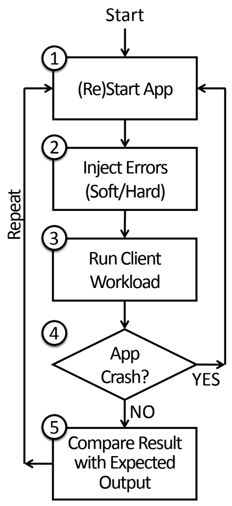


Figure 2: Memory error emulation framework.

[3]Note that we ignore responses to the client that time out due to performance variations (e.g., all of an application's threads are busy serving other requests). In our tests, this occurred for 1.5% of the total requests.

We stop testing after a statistically-significant number of errors have been emulated. While testing all of the memory addresses of data-intensive applications may not be feasible due to their sheer size, in our experiments we ensured that we sampled at least 0.1% of the memory addresses in the applications we examine. This translates to over 10,000 experiments for an application with 32 GB of in-memory data (the average amount of in-memory data for the applications we tested was ~28 GB). We can then process the logged information to quantify an application's tolerance or vulnerability to memory errors (Section III-A).

B. Memory Access Monitoring Framework

To monitor application memory access behavior, we use processor *watchpoints*, available on x86 processors, to execute logging code on individual loads/stores from/to arbitrary memory addresses. Algorithm 1(b) lists the pseudocode for tracking memory access patterns to a random location in an application. In line 1, similar to Algorithm 1(a), we randomly select a valid byte-aligned memory address and store this address in the variable *addr*. In line 2, we register a watchpoint in the debugger (e.g., using `awatch` in `GDB`). From then on, whenever the processor loads data from that address or stores data to that address, our logging code in lines 3–5 will get called to print out the data at *addr*, whether the access was a load or a store (*loadOrStore*), and the time of access (*time*).

We can then process the information logged by Algorithm 1(b), to compute the safe ratio for a region of an application's data (Section III-B). To quantify the recoverability of an application's data (Section III-C), we first use the debugger to identify any read-only file-mapped data, which we classify as recoverable. Additionally, we process the data logged by Algorithm 1(b) to measure how frequently different regions of memory are written to, and classify any regions of memory that are written to every ≥ 5 minutes on average as recoverable (as storing a backup copy of their data in persistent storage would likely not pose a significant performance impediment).

V. Data-Intensive Application Memory Error Tolerance

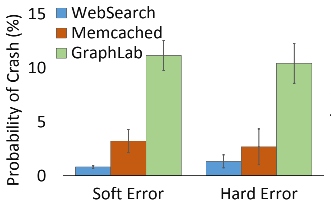
We next use our methodology to quantify the tolerance of three data-intensive applications to memory errors. We use the insights developed in this section to inform the new memory system organizations that we propose in Section VI-B to reduce system cost.

A. Data-Intensive Applications

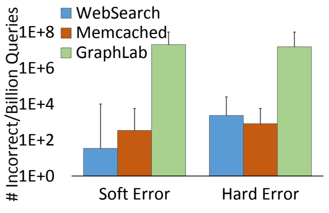
We examined three data-intensive applications as part of our case study: an interactive web search application (WebSearch), an in-memory key–value store (Memcached), and a graph mining framework (GraphLab). While these applications all operate on large amounts of memory (Table 3 lists

Applications	Private	Heap	Stack	Total
WebSearch	36 GB	9 GB	60 MB	46 GB
Memcached	0 GB	35 GB	132 KB	35 GB
GraphLab	0 GB	4 GB	132 KB	4 GB

Table 3: The size of different applications' memory regions.

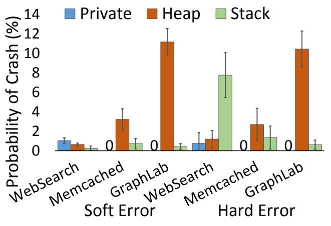


(a) Application vulnerability

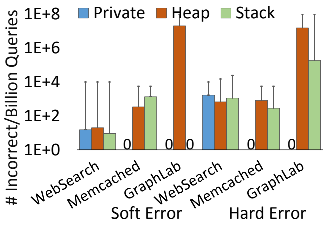


(b) Application incorrectness

Figure 3: Inter-application variations in vulnerability to single-bit soft and hard memory errors for the three applications in terms of (a) probability of crash and (b) frequency of incorrect results.



(a) Memory region vulnerability



(b) Memory region incorrectness

Figure 4: Memory region variations in vulnerability to single-bit soft and hard memory errors for the applications in terms of (a) probability of crash and (b) frequency of incorrect results.

the sizes of their various regions of memory), they differ in several ways that we describe next.

WebSearch implements the index searching component of a production search engine described in [58]. It does this by storing several hundred gigabytes of index data in persistent storage and uses DRAM as a read-only cache for around 36 GB of frequently-accessed data. We use a real-world trace of 200,000 queries as the client workload. The result of a query is a set of the top four most relevant documents to the query. We use number of documents returned, the relevance of the documents to the query, and the popularity score of the documents as the expected outputs.

To evaluate WebSearch, we used 40 Intel Xeon two-socket servers with 64 GB DDR3 memory and conducted our experiments for over two months. In total, we sampled 20,576 unique memory addresses for our experiments across different memory regions, executing a total of over five billion queries.

Memcached [59] is an in-memory key–value store for a 30 GB Twitter dataset. It primarily uses DRAM to improve the performance of read requests. We run a synthetic client workload that consists of 90% read requests and 10% write requests. We use the contents of the value fetched by read requests as the expected outputs.

To evaluate Memcached, we used an Intel Xeon two-socket server with 48 GB DDR3 memory and sampled more than 983 unique memory addresses for our experiments across different memory regions, executing a total of over six billion queries.

GraphLab [60] is a framework designed to perform computation over large datasets consisting of nodes and edges. We use a 1.3 GB dataset of 11 million Twitter users as nodes with directed edges between the nodes representing whether one Twitter user follows another. Our workload runs an algorithm called TunkRank [61] that assigns each node in a graph a floating-point popularity score corresponding to that user's influence. We use the scores of the 100 most influential users as the expected outputs.

To evaluate GraphLab, we used an Intel Xeon two-socket server with 48 GB DDR3 memory and sampled more than 2,159 unique memory addresses for our experiments across different memory regions.

B. Application Characterization Results

We applied our characterization methodology from Section IV to our applications and next present the results and insights from this study.

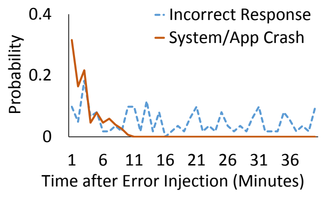
Finding 1: Error Tolerance Varies Across Applications. Figure 3(a) plots the probability of each of the three applications crashing due to the occurrence of single-bit soft or hard errors in their memory (i.e., application vulnerability). Error bars on the figure show the 90% confidence interval of each probability. In terms of how these errors affect application correctness, Figure 3(b) plots the rate of incorrect results per billion queries under the same conditions. Error bars on the figure label the maximum number of incorrect queries during a test per billion queries (the minimum number of incorrect queries during a test per billion was 0). We draw two key observations from these results.

First, there exists a significant variance in vulnerability among the three applications both in terms of crash probability and in terms of incorrect result rate, which varies by up to six orders of magnitude. Second, these characteristics may differ depending on whether errors are soft or hard (for example, the number of incorrect results for WebSearch differs by over two orders of magnitude between soft and hard errors). We therefore conclude that *memory reliability techniques that treat all applications the same are inefficient because there exists significant variation in error tolerance among applications.*

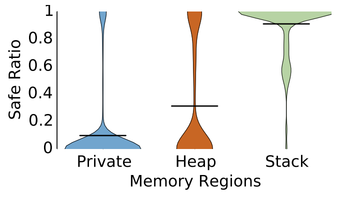
Finding 2: Error Tolerance Varies Within an Application. Figure 4(a) plots the probability of each of the three applications crashing due to the occurrence of single-bit soft or hard errors in different regions of their memory (with error bars showing the 90% confidence interval of each probability). Figure 4(b) plots the rate of incorrect results per billion queries under the same conditions (with error bars showing the maximum number of incorrect queries during a test per billion queries, with the minimum number being 0). Two observations are in order.

First, for some regions, the probability of an error leading to a crash is much lower than for others (for example, in WebSearch the probability of a hard error leading to a crash in the *heap* or *private* memory regions is much lower than in the *stack* memory region). Second, even in the presence of memory errors, some regions of some applications are still able to tolerate memory errors (perhaps at reduced correctness). This may be acceptable for applications such as WebSearch that aggregate results from several servers before presenting them to the user, in which case the likelihood of the user being exposed to an error is much lower than the reported probabilities. We therefore conclude that *memory reliability techniques that treat all memory regions within an application the same are inefficient because there exists significant variance in the error tolerance among different memory regions.*

For the remainder of this section, we use WebSearch as our exemplar data-intensive application and perform an in-depth analysis of its tolerance behavior.



(a) Temporal variation



(b) Safe ratio

Figure 5: (a) Temporal variation in vulnerability for WebSearch, presented as frequency distribution function of time between a memory error is injected and a corresponding effect of that error (either an incorrect result or a crash) is observed. (b) Safe ratio distribution for different memory regions of WebSearch.

Finding 3: Quick-to-Crash vs. Periodically Incorrect Behavior. Figure 5(a) shows the probability of a particular type of outcome occurring after a certain number of minutes given that it occurs during an application's execution time. For this analysis, we focus on soft errors and note that hard errors would likely shift these distributions to the right, as hard errors would continue to be detected over time. We draw two key conclusions from this figure.

First, more-severe failures of an application or system due to a memory error appear to be exponentially distributed and exhibit a *quick-to-crash* behavior: they are detected early-on (within the first ten minutes for WebSearch) and result in an easily-detectable failure mode. Second, less-severe failures of an application due to a memory error appear to be uniformly distributed and exhibit *periodically incorrect* behavior: they arise more evenly distributed over time as incorrect data is accessed.

Finding 4: Some Memory Regions Are Safer Than Others. Recall from Section III-B that the *safe ratio* quantifies the relative frequency of writes to a memory region versus all accesses to the memory region as a way to gauge the likelihood of an error being masked by an overwrite in an application. We sampled the addresses of the memory regions of WebSearch and plot the distribution of their safe ratios in Figure 5(b) (we sampled 1590 addresses in total, with the number of sampled addresses in each memory region roughly proportional to the size of that region). In Figure 5(b), the width of each colored region indicates the probability density for a given safe ratio of the memory region indicated at the bottom. The line on each distribution denotes the average safe ratio value for that region. We make two observations from these results.

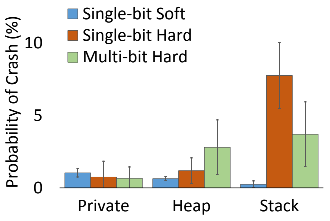
First, we notice a difference in the safe ratios between the regions of the application that contain programmer-managed data, the *private* and *heap* regions, and the region of the application that contains compiler-managed data, the *stack*. The programmer-managed regions (*private* and *heap*), which in WebSearch contain mainly read-only web index data, have a smaller potential to mask memory errors with overwrites due to their lower write intensity. The compiler-managed region (*stack*), on the other hand, contains data that is frequently expanded and discarded whenever new functions are called or returned from, giving it a high potential to mask memory errors with overwrites (cf. Figure 1 outcome 1).

Second, we note that though memory regions may have relatively low safe ratios (such as *private* and *heap*), errors in these regions may still be masked by application logic (cf. Figure 1 outcome 2.1). For example, the *private* and *heap* regions, despite their low safe ratios, are still quite robust against memory errors (with crash probabilities of 1.04% and 0.64%, respectively, in Figure 4a and low rates of incorrect results in Figure 4b). Prior works have examined techniques to identify how applications mask errors due to their control flow [62, 63]. These can potentially be used in conjunction with our technique to further characterize the safety of memory regions.

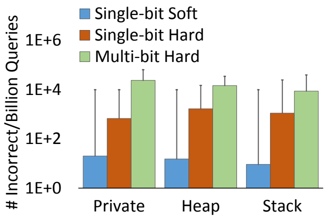
We conclude that *some application memory regions are safer than others because either application access pattern or application logic can be the dominant factor in different memory regions to mask a majority of memory errors.*

Finding 5: More Severe Errors Mainly Decrease Correctness. We next examine how error severity affects application vulnerability. To do so, we emulated three common error types: single-bit soft errors and single-/two-bit hard errors. Figure 6(a) shows the how different error types affect the probability of a crash and Figure 6(b) shows how the different error types affect the rate of incorrect results. We find that while the probability of a crash is relatively similar among the different error types (without any conclusive trend), the rate of incorrect results can increase by multiple orders of magnitude. Especially large is the increase in incorrect result rates from single-bit soft errors to single-bit hard errors, which may vary by around three orders of magnitude. Thus, *more severe errors seem to have a larger effect on application correctness as opposed to an application's probability of crashing*.

Finding 6: Data Recoverability Varies Across Memory Regions. We used our methodology from Section IV-B to measure the amount of memory in WebSearch that was implicitly recoverable (read-only data with a copy on persistent storage) and explicitly recoverable (data that is written to every ≥5 minutes on average) and show our results in Table 5. Note that the same data may be both implicitly and explicitly recoverable, so the percentages in Table 5 may sum to greater than 100%. We make three observations from this data. First, we find that for a significant fraction of the address space (at least 82.1%, pessimistically assuming the implicitly recoverable and explicitly recoverable data completely overlap) of WebSearch, it is feasible to recover a clean copy from disk with low overhead. This suggests that there is ample opportunity for correcting memory errors in software with low overhead. Second, we find that, for



(a) WebSearch vulnerability



(b) WebSearch incorrectness

Figure 6: Vulnerability of WebSearch to types of errors in terms of (a) probability of crash and (b) frequency of incorrect result.

Memory region	Implicitly recoverable	Explicitly recoverable
Private	88%	63.4%
Heap	59%	28.4%
Stack	1%	16.7%
Overall	82.1%	56.3%

Table 5: Recoverable memory in WebSearch.

Design dimension	Technique	Benefits	Trade-offs
Hardware techniques	No detection/correction	No associated overheads (low cost)	Unpredictable crashes and silent data corruption
	Parity	Relatively low cost with detection capability	No hardware correction capability
	SEC-DED/DEC-TED	Tolerate common single-/double-bit errors	Increased cost and memory access latency
	Chipkill [10]	Tolerate single-DRAM-chip errors	Increased cost and memory access latency
	Mirroring [12]	Tolerate memory module failure	100% capacity overhead
	Less-Tested DRAM	Saved testing cost during manufacturing	Increased error rates
Software responses	Consume errors in application	Simple, no performance overhead	Unpredictable crashes and data corruption
	Automatically restart application	Can prevent unpredictable application behavior	May make little progress if error is frequent
	Retire memory pages	Low overhead, effective for repeating errors	Reduces memory space (usually very little)
	Conditionally consume errors	Flexible, software vulnerability-aware	Memory management overhead to make decision
	Software correction	Tolerates detectable memory errors	Usually has performance overheads
Usage granularity	Physical machine	Simple, uniform usage across memory space	Costly depending on technique used
	Virtual machine	More fine-grained, flexible management	Host OS is still vulnerable to memory errors
	Application	Manageable by the OS	Does not leverage different region tolerance
	Memory region	Manageable by the OS	Does not leverage different page tolerance
	Memory page	Manageable by the OS	Does not leverage different data object tolerance
	Cache line	Most fine-grained management	Large management overhead; software changes

Table 4: Heterogeneous reliability design dimensions, techniques, and their potential benefits and trade-offs.

WebSearch, more data is implicitly recoverable than explicitly recoverable. This is likely due to the fact that most of WebSearch's working set of data is an in-memory read-only cache of web document indices stored on disk. Thus, a large portion of the data in applications like WebSearch that cache persistent data in memory can tolerate memory errors in software, without any additional need to maintain a separate copy of data on disk. Third, even if WebSearch did not maintain a copy of its data on disk, the majority of its data (56.3%) could still be recovered with relatively low overhead. We therefore conclude that *for data-intensive applications like WebSearch, software-only memory error tolerance techniques are a promising direction for enabling reliable system designs.*

Based on the results of our case study of data-intensive applications, we next explore how to design systems composed of heterogeneous hardware/software techniques to enable high reliability at low system cost.

VI. Heterogeneous-Reliability Memory Systems

Overview. The goals for our heterogeneous-reliability memory system design methodology are (1) to provide high memory system reliability (e.g., enabling 99.90% single server availability) at (2) low cost. To achieve these goals, our methodology consists of three steps. First, motivated by the fact that memory error *vulnerability* and *recoverability* varies widely among different applications and memory regions, we perform an exploration of the design space for systems that could employ heterogeneous hardware/software reliability techniques. Second, we examine how ways of mapping applications with different reliability characteristics to system configurations in the heterogeneous-reliability design space can achieve the right amount of reliability at low cost, using the WebSearch application as an example. Third, we discuss some of the required hardware/software support for the proposed new system designs. We describe each of these steps in detail next.

A. Design Space, Metrics, and Error Models

We examine three dimensions in the design space for systems with heterogeneous reliability: (1) hardware techniques to detect and correct errors, (2) software responses to errors, and (3) the granularity at which different techniques are used. Table 4 lists the techniques we considered in each of the dimensions along with their potential benefits and trade-offs. In addition to discussing system designs, we discuss (1) the metrics we use to evaluate the benefits and costs of the designs, and (2) the memory error model we use to examine the effectiveness of the designs. We discuss each of these components of our design space exploration in turn.

Hardware Techniques. Hardware techniques to detect and correct errors determine the amount of protection provided by the memory devices in a system. These techniques typically trade hardware cost for additional memory error detection and correction capabilities, and so we would like to use them as sparingly as possible to achieve a particular amount of reliability in our system designs. We briefly review the various techniques next.

One technique employed in consumer laptops and desktops to reduce cost is to not use any memory error detection and correction capabilities in memory. While this technique can save greatly on cost, it can also cause unpredictable crashes and data corruption, as has been documented in studies across a large number of consumer PCs [64]. Other techniques (discussed in Section II) that store additional data in order to detect and/or correct memory errors include storing parity information, SEC-DED, Chipkill, and Mirroring. These more costly techniques are typically employed homogeneously in existing servers to achieve a desired amount of reliability. Another technique that is orthogonal to those we have discussed so far is to employ memory devices that use less-tested DRAM chips. The testing performed by vendors in order to achieve a certain quality of DRAM device can add a substantial amount of cost [8], though it can increase the average reliability of DRAM devices.

Software Responses. Software responses to errors determine any actions that are taken by the software to potentially tolerate or correct a memory error. These techniques typically trade application/OS performance and implementation complexity for improved reliability. Notice that some of these software responses can potentially enable lower-cost hardware techniques to be employed in a system, while still achieving a desired amount of reliability. We discuss each of these techniques next.

The simplest (but potentially least effective) software response to a memory error is to allow the application to consume the error and experience unpredictable behavior or data corruption [65]. As we have seen, however, some data-intensive applications (such as WebSearch) can consume and tolerate a considerable amount of errors without substantial consequences. Another software response is to automatically restart applications once an error has been detected (whether through hardware or software techniques or other indicators, such as an application crash), which can reduce the likelihood that applications experience unpredictable behavior. Retiring memory pages is a technique used in some operating systems [15, 20–22] to help tolerate memory errors by not allowing a physical page that has experienced a certain number of errors to be allocated to applications.

Other software responses may require additional software/OS modification in order to implement. Allowing applications to conditionally consume errors once they have been identified can potentially allow the software/OS to make informed decisions about how tolerable errors in certain memory locations may be. Recent work that allows the programmer to annotate the tolerance of their data structures (e.g., [47]) is one way of allowing software to conditionally consume errors. Finally, the software/OS may be able to perform its own correction of memory errors once they are detected in the hardware by storing additional data required to repair corrupted data. Though these techniques require software changes and may reduce performance, they have the potential to reduce hardware cost by making up for the deficiencies of less reliable, but cheaper, hardware devices.

Usage Granularity. The usage granularity determines where hardware techniques or software responses are applied. Applying hardware techniques or software responses at coarser granularities may require less overhead to manage, but may not leverage the diversity of tolerance among different applications and memory regions. For example, the coarsest granularity, applying techniques across the entire physical machine, is what is typically used in servers, and may be inefficient for applications or memory regions that have different reliability requirements (like the examples we have examined in data-intensive applications).

Finer granularities for using hardware techniques and software responses than the entire physical machine include across individual virtual machines, applications, memory regions/pages, and cache lines. Each increasingly finer granularity of usage provides more opportunities for how different hardware techniques and software responses are used across data, but may also require more complexity to manage the increased number of techniques and responses used.

Metrics. When evaluating a particular system configuration in the design space, we examine three metrics to quantify the effectiveness of a design: (1) *cost*, the amount spent on server hardware for a particular system design, (2) *single server availability*, the fraction of time a server is functioning correctly over a certain period of time,[4] and (3) *reliability*, the likelihood of incorrect behavior occurring.

Error Models. To evaluate the reliability and availability of a particular design, we use information from prior studies (e.g., [13, 15]) to derive the rate of occurrence of soft and hard errors. We then examine how such rates of occurrence affect application reliability and single server availability based on the measured results from our case studies of applications.

[4]Note that this is different from *application* availability, which measures the fraction of time an application is functioning correctly, *typically across multiple servers*. In fact, for our applications, application availability would likely be much *higher* than single *server* availability because our applications rely on software-level techniques that can tolerate server failures.

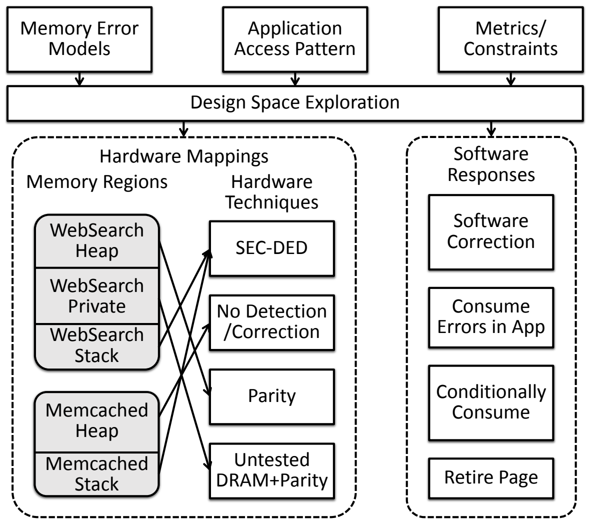


Figure 7: Mapping of applications to a heterogeneous-reliability memory system design, arrows show an example mapping.

Note that though we do not explicitly model how error rates change depending on DRAM density, capacity, speed, power, and other characteristics, our methodology is compatible with other more complex, and potentially more precise, ways of modeling the rate of memory error occurrence.

Tying everything together, Figure 7 illustrates the high level operation of our methodology. At the top of the figure are the inputs to our technique: memory error models, application memory access information (such as an application's spatial and temporal locality of reference), and the various metrics under evaluation and their constraints (e.g., 99.90% single server availability). We choose as our usage granularity memory regions for their good balance between diversity of memory error tolerance and lower management complexity. Based on the inputs, we explore different mappings of memory regions to hardware techniques and which software responses to use. Finally, we choose the heterogeneous-reliability memory system design that best suits our needs. We next examine the results of performing this process on the WebSearch application and the resulting implications for hardware/software design.

B. Design Space Exploration

In this section, we use the results of our case study presented in Section V and the design space parameters discussed in Section VI to evaluate the effectiveness and cost of several memory system design points for the WebSearch application. We chose five design points to compare against to illustrate the inefficiencies of traditional homogeneous approaches to memory system reliability as well as show some of the benefits of heterogeneous-reliability memory system designs, which we next describe and contrast.

- ***Typical Server:*** A configuration resembling a typical server deployed in a modern datacenter. All memory is homogeneously protected using SEC-DED ECC.
- ***Consumer PC:*** Consumer PCs typically have no hardware protection against memory errors, reducing both their cost and reliability.

Design parameters	
DRAM/server HW cost	30%
NoECC memory cost savings	11.1%
Parity memory cost savings	9.7%
L memory cost savings	18%±12%
Crash recovery time	10 mins
Par+R flush threshold	5 mins
Errors/server/month [13]	2000
Target single server availability	99.9%

	Mapping			Metrics				
Configuration	Private (36GB)	Heap (9GB)	Stack (60MB)	Memory cost savings (%)	Server HW cost savings (%)	Crashes/ server/ month	Single server availability	# incorrect/ million queries
Typical Server	ECC	ECC	ECC	0.0	0.0	0	100.00%	0
Consumer PC	NoECC	NoECC	NoECC	11.1	3.3	19	99.55%	33
Detect&Recover	Par+R	NoECC	NoECC	9.7	2.9	3	99.93%	9
Less-Tested (L)	NoECC	NoECC	NoECC	27.1 (16.4-37.8)	8.1 (4.9-11.3)	96	97.78%	163
Detect&Recover/L	ECC	Par+R	NoECC	15.5 (3.1-27.9)	4.7 (0.9-8.4)	4	99.90%	12

Table 6: Mapping of WebSearch address space to different hardware/software design points and the corresponding trade-off in cost, single server availability, and reliability. (ECC = SEC-DED memory; NoECC = no detection/correction; Par+R = parity memory and recovery from disk; L = less-tested memory.)

- ***Detect&Recover:*** Based on our observation that some memory regions are safer than others (Section V), we consider a memory system design that, for the *private* region, uses parity in hardware to detect errors and responds by correcting them with a clean copy of data from disk in software (Par+R, parity and recovery), and uses neither error detection nor correction for the rest of its data.
- ***Less-Tested (L):*** Testing increases both the cost and average reliability of memory devices. This system examines the implications of using less-thoroughly-tested memory throughout the entire system.
- ***Detect&Recover/L:*** This system evaluates the Detect&Recover design with less-tested memory. ECC is used in the *private* region and Par+R in the *heap* to compensate for the reduced reliability of the less-tested memory.

Table 6 (left) shows the design parameters that we use for our study. We estimate the fraction of DRAM cost compared to server hardware cost based on [6]. We derive the cost of ECC DRAM, non-ECC DRAM (NoECC), and parity-based DRAM using "Added capacity" from Table 1. We estimate the cost of less-tested DRAM based on the trends shown in [8, 9] and examine a range of costs for less-tested DRAM because we are not aware of any recently documented costs from vendors. Crash recovery time is based on our observations during testing and we assume that data that is written in memory for regions protected by parity and recovery (Par+R) is copied to a backup on disk every five minutes. We assume 2000 errors per server per month based on a prior study of DRAM errors in the field [13] and target single server availability of 99.90%. Since we do not examine the occurrence of hard errors in our analysis (only their ongoing effects), we treat all memory errors as soft errors for this analysis.

Table 6 (right) shows, for each of the five designs we evaluate, how their memory regions are mapped to different hardware/software reliability techniques (columns 2–4), and the resulting effects on the metrics that we evaluate (columns 5–9). We break down cost savings into two components: the cost savings of the memory system (column 5) and the cost savings of the entire server hardware (column 6). Note that we compute capital cost as it represents the dominant cost component of the server.[5] We also list the average number of crashes per server per month (column 7) and the associated fraction of single server availability (column 8). While server availability also depends on other factors, such as disk failure or power supply failure, here, we examine server availability from the perspective of only memory errors. We also show the number of incorrect responses per million queries (column 9).

We take the Typical Server configuration as our baseline. Notice that the memory design common in consumer PCs, which uses no error detection or correction in memory, is able to reduce memory cost by 11.1% (reducing server hardware cost by 3.3%). This comes at the penalty of 19 crashes per server per month (reducing single server availability to 99.55%). When the server is operational, it generates 33 incorrect results per million queries.

In contrast, the Detect&Recover design, which leverages a heterogeneous-reliability memory system, is able to reduce memory cost by 9.7% compared to the baseline (leading to a server hardware cost reduction of 2.9%), while causing only 3 crashes per server per month (leading to a single server availability of 99.93%, above the target availability of 99.90%). In addition, during operation, the Detect&Recover design generates only 9 incorrect results per million queries.

Using less-tested DRAM devices has the potential to reduce memory costs substantially—in the range of 16.4% to 37.8% (leading to server hardware cost reduction in the range of 4.9% to 11.3%). Unfortunately, this reduction in cost comes with a similarly substantial reduction in server availability and reliability, with 96 crashes per server per month, a single server availability of only 97.78%, and 163 incorrect queries per million.

The Detect&Recover/L technique uses a heterogeneous-reliability memory system on top of less-tested DRAM to achieve both memory cost savings and high single server availability/reliability. While using less-tested DRAM lowers the cost of the memory by 3.1% to 27.9% (reducing cost of server hardware by 0.9% to 8.4%), using hardware/software reliability techniques tailored to the needs of each particular memory region reduces crashes to only 4 per server per month (meeting the target single server availability of 99.90%), with only 12 incorrect queries per million. We therefore conclude that *heterogeneous-reliability memory system designs can enable systems to achieve both high cost savings and high single server availability/reliability at the same time*.

While we performed our analysis on the WebSearch application, we believe that other data-intensive applications may benefit from the heterogeneous-reliability memory system designs we have proposed. To illustrate this fact, Figure 8 shows for each data-intensive application we examined in Section V, the maximum number of tolerable errors required to still achieve a particular amount of reliability,

[5]We also expect memory system energy savings to be proportional to memory capacity savings. This is due to the eliminated dynamic energy of moving and storing the ECC bits used for error detection or correction.

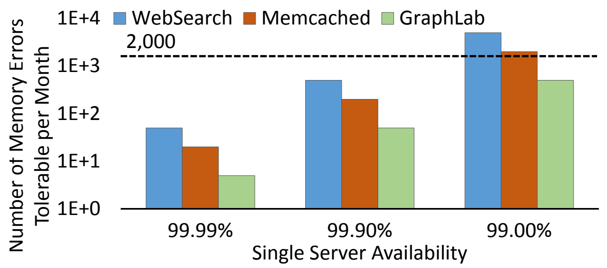


Figure 8: Number of memory errors tolerable per month to ensure different single server availability requirements are met for three applications.

derived from the results in Figure 4. We make two observations from this figure. First, even at the error rate of 2000 errors per month that we assume, without any error detection/correction, two of the applications (WebSearch and Memcached) are able to achieve 99.00% single server availability. Second, there exists an order of magnitude difference in the amount of errors that can be tolerated per month to achieve a particular availability across the applications. Based on these, we believe that there is significant opportunity in these and other applications for reducing server hardware cost while achieving high single server availability/reliability using our heterogeneous-reliability design methodology.

C. Required Hardware/Software Support

We next discuss the hardware and software changes required to enable heterogeneous-reliability memory systems and the feasibility of foregoing memory error detection/correction in servers.

Hardware Support. In hardware, our techniques require the ability for memory modules with different hardware reliability techniques to coexist in the same system (e.g., no detection/correction, parity detection, and SEC-DED). We believe that this is achievable using existing memory controllers at the granularity of memory channels without much modification. Figure 9 shows an example of how such a technique could be employed on a processor that uses separate memory controllers for each channel, each controlling DIMMs that employ different hardware reliability techniques. Furthermore, techniques such as memory disaggregation [66] can be leveraged to provide heterogeneous-reliability memory over remote direct memory access protocols to a variety of servers.

Software Support. In software, our techniques require the ability to identify memory regions with distinct reliability requirements and map them to locations on hardware devices employing appropriate reliability techniques. Although we examined OS-visible memory regions in this work (*private*, *heap*, and *stack*), our technique can easily be extended to other usage granularities, potentially by leveraging hints from the programmer (as in [47, 67]). Alternatively, it is foreseeable with our technique that infrastructure service providers, such as Amazon EC2 and Windows Azure, could provide different reliability domains for users to configure their virtual machines with depending on the amount of availability they desire (e.g., 99.90% versus 99.00% availability). In addition, techniques like virtual machine migration [68] can be used to dynamically change the provided memory reliability over time (e.g., in response to changes in application utilization).

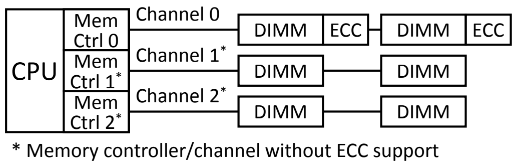


Figure 9: Minimal changes in today's memory controller can achieve heterogeneous memory provisioning at the channel granularity.

Feasibility. A primary concern associated with using memory without error detection/correction is the fear of the effects of memory errors propagating to persistent storage. Based on our analysis, we believe that, in general, the use of memory without error detection/correction (and the subsequent propagation of errors to persistent storage) is suitable for applications with the following two characteristics: (1) application data is mostly read-only in memory (i.e., errors have a low likelihood of propagating to persistent storage), and (2) the result from the application is transient in nature (i.e., consumed immediately by a user and then discarded). There are several large-scale data-intensive applications that conform to these characteristics such as web search (which we have analyzed), media streaming, online games, collaborative editing (e.g., Wikipedia), and social networking.

Applications that do not have these two characteristics should be carefully inspected and possibly re-architected before deciding to use memory without detection/correction capabilities. As an example, applications could divide their data into persistent and transient, and map persistent data to more reliable memory. To avoid incorrect results from remaining in memory indefinitely, applications may periodically invalidate and recompute their working set of data.

Another concern with using less reliable memory devices is their inability to detect (and thus retire [20–22]) memory pages with permanent faults. To alleviate this issue, software designed to detect memory errors, such as `memtest` [69], can be run periodically on servers with memory devices without error detection and pages can be retired as usual.

Finally, the use of less reliable memory may result in single server unavailability, and hence, may only be applicable for applications that possess inherent slack in their availability requirements. Server models such as scale-out and stateless models, which tolerate single server failures by diverting work to other available servers, are well-suited to handle occasional single server unavailability, and thus can even more efficiently leverage heterogeneous-reliability memory system designs.

VII. Conclusions

In this paper, we developed a new methodology to quantify the tolerance of applications to memory errors. Using this methodology, we performed a case study of three new data-intensive workloads that showed, among other new insights, that there exists a diverse spectrum of memory error tolerance both within and among these applications. We proposed several new hardware/software heterogeneous-reliability memory system designs and evaluated them to show that the one-size-fits-all approach to reliability in modern servers is inefficient in terms of cost, and that heterogeneous-reliability systems can achieve the benefits of both low cost and high single server availability/reliability. We hope that our techniques can enable the use of lower-cost memory devices to reduce the server hardware cost of datacenters and that our analyses will spur future research on heterogeneous-reliability memory systems. As DRAM technology scales into small feature sizes and becomes less reliable, we believe such system-level hardware/software cooperative heterogeneous-reliability solutions will become more important in the future [70].

As part of our future efforts, we plan to extend our characterization framework to cover a more diverse set of memory failure modes (e.g., failures correlated across DRAM banks, rows, and columns), and to explore lighter-weight characterization methodologies to make characterizing application memory error tolerance cheaper for a wider set of applications. We also intend to implement and further evaluate the heterogeneous hardware detection and software recovery designs we propose in this paper.

VIII. Acknowledgments

We thank the anonymous reviewers and the members of SAFARI research group for feedback. We acknowledge the support of Microsoft and Samsung. This research was partially supported by grants from Intel Science and Technology Center for Cloud Computing, NSF Awards 0953246, 1065112, 1212962. This research was started as an internship project at Microsoft Research.

References

[1] L. A. Barroso *et al.*, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.
[2] E. M. Elnozahy *et al.*, "Energy-Efficient Server Clusters," in *PACS*, 2003.
[3] P. Jobin, "Cloud Computing Shifting to Cooler Climates," 2012, http://tinyurl.com/mfrlrtl.
[4] S. Grundberg *et al.*, "For Data Center, Google Goes for the Cold," 2011, http://tinyurl.com/ml55nh5.
[5] A. C. Riekstin *et al.*, "No More Electrical Infrastructure: Towards Fuel Cell Powered Data Centers," in *HotPower*, 2013.
[6] C. Kozyrakis *et al.*, "Server Engineering Insights for Large-Scale Online Services," *IEEE Micro*, 2010.
[7] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *NSDI*, 2013.
[8] Z. Al-Ars, "DRAM Fault Analysis and Test Generation," Ph.D. dissertation, Delft, 2005.
[9] "Memory Test Background," 2000, http://tinyurl.com/m7c3wf7.
[10] T. J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," *IBM Microelectronics Division*, 1997.
[11] P. J. Meaney *et al.*, "IBM zEnterprise Redundant Array of Independent Memory Subsystem," *IBM JRD*, 2012.
[12] D. Henderson *et al.*, "POWER7 System RAS," 2012.
[13] B. Schroeder *et al.*, "DRAM Errors in the Wild: A Large-Scale Field Study," in *SIGMETRICS Performance*, 2009.
[14] V. Sridharan *et al.*, "A Study of DRAM Failures in the Field," in *SC*, 2012.
[15] A. A. Hwang *et al.*, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," in *ASPLOS*, 2012.
[16] X. Li *et al.*, "A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility," in *USENIX ATC*, 2010.
[17] J. Stuecheli *et al.*, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *MICRO*, 2010.
[18] JEDEC Solid State Technology Association, "JEDEC Standard: DDR3 SDRAM, JESD79-3C," 2008.
[19] D. Fiala *et al.*, "Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing," in *SC*, 2012.
[20] "Predictive Failure Analysis (PFA)," http://tinyurl.com/n34z657.
[21] "Mcelog: Memory Error Handling in User Space," http://www.halobates.de/lk10-mcelog.pdf.
[22] D. Tang *et al.*, "Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults," in *DSN*, 2006.
[23] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[24] D. Lee *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[25] V. Seshadri *et al.*, "RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data," in *MICRO*, 2013.
[26] S. Khan *et al.*, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
[27] Y. Kim *et al.*, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
[28] T. C. May *et al.*, "Alpha-Particle-Induced Soft Errors in Dynamic Memories," *IEEE T-ED*, 1979.
[29] V. Sridharan *et al.*, "Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults," in *SC*, 2013.
[30] T. Siddiqua *et al.*, "Analysis and Modeling of Memory Errors from Large-scale Field Data Collection," in *SELSE*, 2013.
[31] A. Messer *et al.*, "Susceptibility of Commodity Systems and Software to Memory Soft Errors," *IEEE TC*, 2004.
[32] D. Li *et al.*, "Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool," in *SC*, 2012.
[33] X. Li *et al.*, "Application-Level Correctness and Its Impact on Fault Tolerance," in *HPCA*, 2007.
[34] H. Esmaeilzadeh *et al.*, "Architecture Support for Disciplined Approximate Programming," in *ASPLOS*, 2012.
[35] J. Bornholt *et al.*, "Uncertain<T>: A First-order Type for Uncertain Data," in *ASPLOS*, 2014.
[36] "Intel iACT," http://www.github.com/IntelLabs/iACT.
[37] D. H. Yoon *et al.*, "Virtualized and Flexible ECC for Main Memory," in *ASPLOS*, 2010.
[38] H. Schirmeier *et al.*, "RAMpage: Graceful Degradation Management for Memory Errors in Commodity Linux Servers," in *PRDC*, 2011.
[39] C. Borchert *et al.*, "Generative Software-Based Memory Error Detection and Correction for Operating System Data Structures," in *DSN*, 2013.
[40] K. Pattabiraman *et al.*, "Samurai: Protecting Critical Data in Unsafe Languages," in *EuroSys*, 2008.
[41] J. Chang *et al.*, "Automatic Instruction-Level Software-Only Recovery," in *DSN*, 2006.
[42] A. Benso *et al.*, "A C/C++ Source-to-Source Compiler for Dependable Applications," in *DSN*, 2000.
[43] L. Leem *et al.*, "ERSA: Error Resilient System Architecture for Probabilistic Applications," in *DATE*, 2010.
[44] M.-L. Li *et al.*, "Trace-Based Microarchitecture-level Diagnosis of Permanent Hardware Faults," in *DSN*, 2008.
[45] X. Xu *et al.*, "Understanding Soft Error Propagation Using Efficient Vulnerability-Driven Fault Injection," in *DSN*, 2012.
[46] M.-L. Li *et al.*, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *ASPLOS*, 2008.
[47] S. Liu *et al.*, "Flikker: Saving DRAM Refresh-Power Through Critical Data Partitioning," in *ASPLOS*, 2011.
[48] D. H. Yoon *et al.*, "BOOM: Enabling Mobile Memory Based Low-power Server DIMMs," in *ISCA*, 2012.
[49] K. T. Malladi *et al.*, "Towards Energy-proportional Datacenter Memory with Mobile DRAM," in *ISCA*, 2012.
[50] M. K. Qureshi *et al.*, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.
[51] H. Yoon *et al.*, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.
[52] S. Phadke *et al.*, "MLP Aware Heterogeneous Memory System," in *DATE*, 2011.
[53] N. Chatterjee *et al.*, "Leveraging Heterogeneity in DRAM Main Memories to Accelerate Critical Word Access," in *MICRO*, 2012.
[54] J. Meza *et al.*, "Enabling efficient and scalable hybrid memories using fine-granularity dram cache management," *IEEE Computer Architecture Letters*, 2012.
[55] "Windows Debugging," http://tinyurl.com/l6zsqzv.
[56] "GDB: The GNU Project Debugger," http://www.sourceware.org/gdb/.
[57] J. Liu *et al.*, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
[58] V. J. Reddi *et al.*, "Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency," in *ISCA*, 2010.
[59] "Memcached," http://memcached.org/.
[60] Y. Low *et al.*, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *PVLDB*, 2012.
[61] D. Tunkelang, "A Twitter Analog to PageRank," 2009, http://tinyurl.com/9byt4z.
[62] S. K. S. Hari *et al.*, "Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults," in *ASPLOS*, 2012.
[63] N. J. Wang *et al.*, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," in *DSN*, 2004.
[64] E. B. Nightingale *et al.*, "Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs," in *EuroSys*, 2011.
[65] M. C. Rinard *et al.*, "Enhancing Server Availability and Security Through Failure-Oblivious Computing," in *OSDI*, 2004.
[66] K. Lim *et al.*, "Disaggregated Memory for Expansion and Sharing in Blade Servers," in *ISCA*, 2009.
[67] A. Sampson *et al.*, "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," in *PLDI*, 2011.
[68] Y. Du *et al.*, "A Rising Tide Lifts All Boats: How Memory Error Prediction and Prevention Can Help with Virtualized System Longevity," in *HotDep*, 2010.
[69] "Memtest86+," http://www.memtest.org/.
[70] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *MEMCON*, 2013.