COMeT: Continuous Online Memory Test

Musfiq Rahman, Bruce R. Childers and Sangyeun Cho
Computer Science Department, University of Pittsburgh, Pittsburgh PA 15260 USA
{musfiq,childers,cho}@cs.pitt.edu

***Abstract*—Today's computers have gigabytes of main memory due to improved DRAM density. As density increases, smaller bit cells become more susceptible to errors. With an increase in error susceptibility, the need for memory resiliency also increases. Self-testing of memory health can proactively check for errors to improve resiliency. This paper describes a software-only self-test to continuously test memory. We present the challenges and design for an approach, called Continuous Online Memory Testing (COMeT), that targets chip multiprocessors. COMeT tests memory health simultaneously with application execution in anticipation of allocation requests. The approach guarantees that memory is tested within a fixed time interval to limit exposure to lurking errors. We developed and evaluated an implementation of COMeT. On the SPEC CPU2006 benchmarks, COMeT has a low 4% average performance overhead. When emulated errors were injected into physical memory, applications executed 1.13x to 4.41x longer with COMeT than without it.**

***Index Terms*—Self testing; errors; resilience; memory**

I. Introduction

For several decades, DRAM has remained the best choice for main memory due to its relatively low cost, aggressive scalability, low power and good performance. Continued decreases in bit cell size have led to exceptionally large main memories–a laptop can be purchased with several gigabytes, while a server can easily have tens of gigabytes. The massive capacity enabled by DRAM has had a direct consequence on the applications, execution models and processors employed today. Applications can be constructed that operate on massive datasets, memory-intensive execution paradigms, including virtualization and managed run-times, are feasible, and the memory capacity required for chip multiprocessors (CMPs) that run several simultaneous applications can be met.

While the relentless scaling of DRAM has been a key enabler, it also has a down side. As cell size and operating voltage are decreased, the memory becomes more susceptible to errors. Background radiation may lead to more "single event upsets" (i.e., an independent random bit flip) in a small cell size. Other error types, such as interdependent multi-bit transient and hard errors, are also possible, and indeed probable, at the extreme scales used in high-density DRAM. With new materials, smaller device geometry, especially in future node sizes of less than 30*nm*, and increased process variations, these error types may become as important as single event upsets, particularly at operating margins [4, 10, 26].

A two and half year study of reliability in Google's data centers revealed that nearly one third of their machines and 8% of memory modules–using today's memory technology–experienced at least one memory error [23]. The study suggested that the errors were inter-related, and thus, unlikely caused by independent bit flips. This result is even more surprising given that memory with uncorrectable errors was quickly replaced. An overarching conclusion of the study was resiliency techniques are necessary to successfully manage memory reliability. These techniques will be vital as DRAM node size is further decreased.

To help address memory resiliency, this paper presents an online memory self-test, Continuous Online Memory Testing (COMeT), inspired by conventional standalone memory testers (e.g., Memtest86+ [1] and PC BIOS Power-on Self Test). Traditional testers repeatedly do write, read and verify operations on memory locations. Different bit-patterns are written and read back to check for the potential and presence of various error types, including multi-bit errors. The advantage to these techniques is they can stress memory with many patterns to find marginal locations that have errors only in particular situations. Memory testers are used during manufacture, boot up, or machine malfunction (for diagnosis). These conventional approaches usually operate in a non-transparent and offline setting. Instead, COMeT is designed to monitor memory health in a deployed system that is actively executing user applications. Because of its online setting, COMeT must minimize its disruption, especially on performance, and remain transparent to user applications.

COMeT is a diagnostic to check memory health and discover errors [6, 9, 13], rather than a fault-tolerance scheme (e.g., memory error correction [2, 7, 31], checkpointing and rollback [17, 25]). It exercises the entire memory system, including the memory controller(s), memory interconnect, DIMMs, and associated glue logic. COMeT is similar in spirit to memory scrubbing for hardware ECC [16, 20, 21], that periodically reads and writes memory to reset single event upsets. However, COMeT's primary purpose is different: It stresses the memory subsystem to expose marginal locations, rather than avoiding the accumulation of soft errors. COMeT is also implemented purely in software as a kernel process that applies test patterns–i.e., a march test [28, 30]–on physical memory pages to detect unhealthy locations. When a page with an error is detected, the page is retired to provide assurance about memory reliability and forestall the immediate replacement of "bad" DRAM modules.

Because memory test frequency influences how quickly health problems are discovered, COMeT maximizes and guarantees a test rate. This guarantee bounds the exposure, i.e.,

vulnerability, of an application to a memory error. COMeT also handles the realities of a live system: applications have varying memory demand and usage patterns and memory testing must be done on physical resources. The approach targets chip multiprocessors. It uses an idle core to test memory pages before they are requested, which helps mitigate the performance impact of the diagnostic. It also handles memory pages that are long held by an application by proactively migrating the pages to recently tested ones based on its guarantee about vulnerability.

We developed an implementation of COMeT for the Linux CMP memory allocator. We found the implementation imposes a modest 4% average performance overhead in a small-scale server. In an experiment that injected errors in physical memory, COMeT allowed programs to execute 1.13x (200 errors) to 4.41x (1,600 errors) longer than without testing.

The requirements for COMeT (i.e., transparent operation in a live system, low performance overhead, and bounded memory error vulnerability) pose important questions about its design, operation, and implementation. This paper answers these questions, including: 1) what is an appropriate design that is both performance efficient and can guarantee that memory health is regularly checked; 2) how should COMeT be implemented and integrated in an actual system; and 3) is a software-only memory diagnostic feasible from an implementation and performance perspective and can it improve memory resiliency? Based on the answers, we make the contributions:

- The design of a software-only process to online and continuously test main memory's health;
- Techniques to test memory ahead of allocation in a CMP and adaptively adjust test rate to minimize overhead, while achieving a guaranteed bound on the maximum time between successive tests of a page;
- A description of how COMeT can be structured and integrated with an OS kernel; and,
- Evaluation of performance, energy, and resiliency to memory errors, including an analysis of important design and configuration choices.

This paper is organized as follows. Section II describes Continuous Online Testing. Section III discusses how to incorporate COMeT in the OS and Section IV presents an evaluation of our approaches. Section V describes related work and Section VI concludes the paper.

II. Continuous Online Memory Testing

Continuous Online Memory Testing is a software process that checks for errors in physical memory. Memory pages with detected errors can be retired, scrubbed or possibly salvaged for small kernel buffers [29]. By avoiding pages with errors, especially ones with errors that are uncorrectable by hardware mechanisms (e.g., multi-bit errors for SECDED), reliability is improved. Errors can manifest themselves at any point, which necessitates that physical pages are constantly checked. COMeT sweeps through memory at regular intervals with a march test [28, 30]. Multiple march tests can be used with varying periodicity to check for different error types.

A. Observations Influencing COMeT

Although conceptually simple, there are numerous ways that memory testing can be structured and integrated as an online and software-only method. Four observations influenced our design and implementation:

1) The frequency at which memory pages are tested impacts the likelihood that an application will encounter a page with an error—the more recent an actively used page is tested, the less likely an application will hit an error on the page. Thus, the amount of time between successive tests on the same physical page determines a "vulnerability window" during which a memory access could suffer an error. To test the entire memory capacity can be time consuming, which can lead to a long vulnerability window. However, vulnerability window should be minimized.
2) An application is vulnerable only to errors on pages that are used. Pages that are allocated, or will be allocated in the near future, must be tested, but unused ones do not have to be checked. Thus, depending on memory utilization, only a portion of physical memory actually needs to be tested. By checking a smaller number of used pages, test frequency can be increased to reduce the vulnerability window.
3) Applications often have high page turnover. A page may be requested, allocated and then released quickly. After a page is released, the vulnerability window for that page may not have been reached yet. Thus, a page that has been recently tested and returned does not have to be tested again until the guaranteed duration of the vulnerability window is reached. Some pages may be held for a long period and must be tested while used.
4) Test rate only partially determines the actual error exposure of an application. An error may appear shortly after a page is tested, but before the page is tested again. A program will be corrupted only if the error location is accessed and the "bad value" is propagated to sensitive state [10, 11, 14, 15]. Thus, even a modest limit on the vulnerability window duration can be effective.

B. Design of COMeT

Based on these observations, one way to do memory testing is "on demand". An on-demand strategy can reduce the amount of testing by focusing on pages that are actually allocated. A physical page is tested on-demand when it is allocated and mapped to a virtual page. A page that is held for longer than the vulnerability window is periodically tested by migrating the page to one that is tested.

However, an on-demand strategy is naïve: It introduces large performance overhead since it is inherently sequential. In particular, on a page allocation, the amount of time spent on testing a page is fully observed because an application is paused while waiting for its allocation request to be satisfied. Similarly, when page migration is done sequentially with program execution, the program has to wait. Although some

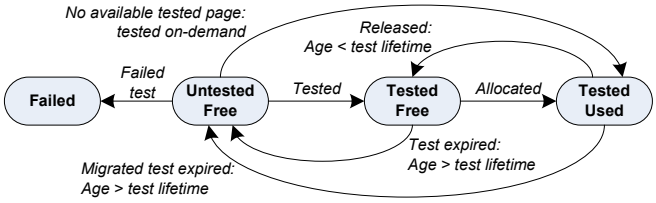


Fig. 1: Page State Diagram

migration latency can be masked by testing and copying pages while an application is blocked, this process is too unpredictable to guarantee the vulnerability window.

A less time consuming and more predictable approach can check page health *concurrently* to program execution in *anticipation* of allocation. With available idleness in a typical commodity CMP, a free core (or cores) can be used to constantly test memory from which allocation requests are satisfied. Migration can be used to copy long-held pages to ones that are already tested before migration begins. This "ahead-of-time" COMeT strategy is an anticipatory one—it plans for allocation by executing march tests on pages before requests.

1) Operation: The key to COMeT is the maintenance of a pool of *unused tested pages* from which memory allocation requests are satisfied. The allocation requests may come from an application's memory usage patterns or the test processes employed by COMeT. The tested pages in the pool must be regularly replenished (i.e., newly tested pages are added as pages are removed or grow old). This strategy cooperatively works with the OS kernel's memory allocator to provide tested pages and gather feedback about memory demand to adjust the rate at which pages in the pool are replenished.

COMeT guarantees that every page used by an application has had its health checked within a fixed time interval. To express this guarantee, we define $age(p)$ as the amount of time elapsed since physical page p was last tested. COMeT guarantees $age(p)+\delta < test\_lifetime$ for all pages p used by an application. δ is a small constant to allow COMeT to check for page expiration prior to it happening. In this way, a bound, determined by $test\_lifetime$, is placed on the vulnerability window and assurances can be made about memory health.

To understand COMeT, first consider a simplified memory allocator. Assume this allocator has a single free list and maintains a page in one of two states – *free* or *used*. A *free* page is on the free list and available for allocation. Physical memory pages are transitioned between these states based on allocation and de-allocation requests. COMeT adds states and transitions to the simplified allocator's state diagram to track whether a page is tested or untested. Figure 1 shows COMeT's state diagram. $age(p)$ is the amount of time that page p spends in a *tested* state (free or used). Initially, p is *untested free*. At some point, p is selected to be tested and transitioned to *tested free* or *failed*. On a successful test, p is moved to *tested free* and remains in this state until it is allocated or $age(p)+\delta \geq test\_lifetime$. A page that does not pass the test is retired and put in the *failed* state.

When a *tested free* page p is requested, it is transitioned to *tested used*. However, it is possible for p to remain in *tested free* long enough that $age(p)+\delta \geq test\_lifetime$, which we call "test expiration". To maintain its guarantee, COMeT must inspect p's age to ensure the page has not expired. If p expired, then it is transitioned to *untested free* to refresh its test. A different unexpired *tested free* page, q, is selected instead to satisfy the memory request.

Similarly, a *tested used* page p can also expire. In this case, COMeT should replace p with a *tested free* page q before p expires. p is migrated to q, and the states of p and q are updated to *untested free* and *tested used*, respectively. COMeT must periodically check p's age while the page is in use. Thus, the transition from *tested used* to *untested free* happens on a regular time interval during application execution. The migration is done concurrently on a separate core; a page under migration cannot be used.

Finally, a transition happens from *untested free* to *tested used* when there are not enough pages in *tested free* to satisfy a memory request. It causes a page to be tested on-demand at the moment of a request, incurring overhead. COMeT tries to avoid this transition maintaining enough *tested free* pages to meet instantaneous demand.

2) Test Guarantee and Replenishment: COMeT relies on two rates. The first rate, termed the "guarantee rate", controls how often to check whether *tested used* pages expire. The second rate—the "replenishment rate"—determines how often to replenish *tested free* pages.

The **guarantee rate** is determined by $test\_lifetime$. To maintain $age(p)+\delta < test\_lifetime$, the rate must be at least $\frac{1}{test\_lifetime-\delta}$. This rate represents only how often COMeT needs to check whether *tested used* pages should be migrated and released. Pages that are *tested free* can have their age checked at time of allocation, as previously described. To set the guarantee rate, $test\_lifetime$ has to be determined. A simple strategy based on physical memory capacity and utilization can be used:

$$test\_lifetime = \alpha \times \frac{\left(\frac{mem\_cap}{page\_size}\right) \times mem\_util \times test\_latency}{test\_res}$$

In this equation, $\alpha \geq 1$ is an adjustment factor set by the system administrator to scale $test\_lifetime$ according to system needs, such as how much power and performance loss due to testing can be tolerated, or even the expected error rate. A larger power or performance budget means a higher test rate can be used. A low error rate would normally permit a slower test rate than the one solely established by memory capacity.

mem_cap is the memory capacity and $page\_size$ is the memory page size. These parameters reflect the physical memory configuration. mem_util is the average percentage of used memory. $mem\_util = 1.0$ sets the guarantee rate conservatively enough that the entire capacity can be tested. However, the whole memory may not be fully utilized (i.e., all pages in *tested used*) at any moment, and the guarantee rate could be set higher by adjusting mem_util. The slack from less than peak utilization can alternatively be exploited to reduce the performance and energy cost of checking memory health since fewer pages have to be checked in unit time.

$test\_latency$ is the amount of time needed to test one page. This latency depends on what tests are done. For example, a comprehensive march test that makes multiple passes over physical memory to read and write different bit patterns could take upwards of $1ms$ per page (there is no caching!). $test\_res$ influences the test rate—the more computational time that COMeT is allowed for testing, the faster the guarantee rate. In a "core rich environment", we may be able to dedicate a core(s) to test memory and set $test\_res = 1.0$.

Unlike the guarantee rate, the **replenishment rate** does not effect the vulnerability of an application to errors. Instead, it determines how often the pool of tested unused pages is filled. Its purpose is to avoid the "demand transition" in the page state diagram. To replenish the pool, a set of *untested free* pages need to be acquired, a march test done, and the pages returned. When the pages are released, they are put in the *tested free* state. The page tests are done concurrently to application execution. The higher the replenishment rate, the more likely that an application request can be satisfied with *tested free* pages and the on-demand transition in Figure 1 can be avoided. However, a high replenishment rate puts more load on the memory subsystem. It also removes more pages on average from the memory allocator, impacting the allocator's ability to satisfy requests in overload situations. It also causes the allocator's performance to suffer due to locking and internal bookkeeping (e.g., breaking buddy blocks into smaller ones). If the replenishment rate is set too low, there may not be enough *tested free* pages available, which can cause overhead due to the on-demand transition.

COMeT walks the tight rope between a high and a low replenishment rate by monitoring memory requests to adjust the rate. As described later, our implementation uses an adaptive strategy based on recent history of memory requests to determine a current replenishment rate. The rate by itself is insufficient since it can be satisfied in different ways. Two parameters determine the replenishment rate: $block\_demand$ and $replenish\text{-}\delta$. $block\_demand$ is number of pages to test every interval of $replenish\text{-}\delta$ time. COMeT dynamically determines $block\_demand$, but $replenish\text{-}\delta$ is statically fixed.

III. COMeT Implementation

We implemented the ahead of time COMeT strategy in Linux to examine how the approach can be integrated with a modern operating system.

A. Architecture

Figure 2 shows COMeT's implementation, which adds several timers and threads to the Linux kernel. As the figure shows, memory allocation is partitioned in Linux among global and per-CPU page frame cache (PCP) allocators. Each allocator has free lists that hold blocks of continguous unused pages. The global allocator uses buddy lists, where a free list is an *order* of rank i that holds free blocks of size 2^i. There are orders $0 \leq i \leq 10$ in Linux. For PCP allocation, there are separate free lists (hot and cold) for each core, which locally cache free blocks of size 1. The allocators have separate free

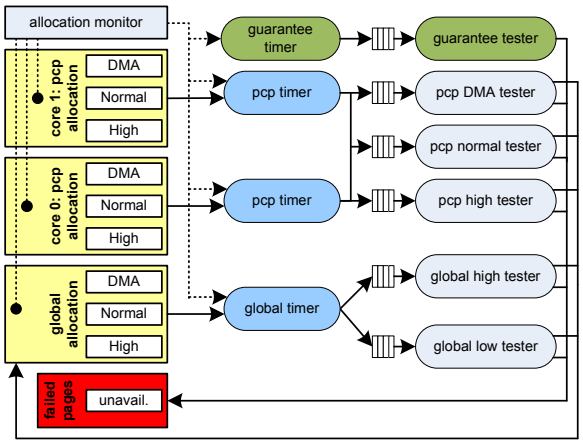


Fig. 2: Architecture of COMeT Implementation in Linux

lists for different memory zones (DMA, normal and high). Allocation is hierarchical: the PCP allocator is tried first. If this fails, then the global allocator is tried.

Most changes made to Linux for COMeT are independent to the actual memory allocators, with three exceptions. First, we add interfaces to the memory allocators to request untested blocks and insert tested blocks into free lists. These interfaces can be invoked only by kernel threads. Second, we change the memory allocators to gather and expose information about memory request demand. Finally, the allocators are changed to return a tested block for an user allocation request. If a tested block is unavailable, then the allocator invokes a march test on a block to satisfy the request. This change implements the demand transition from Figure 1.

COMeT introduces new kernel timers to guarantee the vulnerability window and to replenish the free lists. Figure 2 shows four timers: one for the vulnerability window guarantee ("guarantee timer"), one for global allocation ("global timer"), and two for per-core allocation on each core ("pcp timer"). When the guarantee timer expires, physical pages are checked for test expiration. When an allocation timer (global or PCP) expires, its handler pulls untested blocks, based on monitored demand, from the appropriate free lists to be tested.

New kernel threads are also added to do the actual testing (e.g., "pcp normal tester" and "global low order tester"). The timer handlers pass memory blocks to these threads for testing. The testers check the blocks, and return blocks to the memory allocators. When a page is discovered that has an error, a tester puts the page into a fault map of failed pages. The test threads are arranged by zone and rank. This organization mirrors Linux's strategy for a CMP, which localizes testing and permits different thread priorities based on latency and importance to satisfying memory requests.

B. Allocation Monitor

The allocation monitor holds a buffer of counters to track allocation events. The counters record memory demand over the time interval $replenish\text{-}\delta$. Each free list has its own counters because demand varies per list. There are two counters for each list: $allocated$ counts the number of blocks allocated and $tested$ counts number of free tested blocks. $allocated$

```
GUARANTEE-HANDLER()
1  TimeStamp time = GET-TIME()
2  PageList expired = FIND-EXPIRED-PAGES(time)
3  foreach Page page in expired
4    Page newpage = ALLOCATE(0)
5    MARK-PTE-MIGRATING(page)
6    COPY(newpage, page)
7    UPDATE-PTE(newpage, page)
8    RELEASE(page,0)
9  SET-TIMER(this, δ − (GET-TIME() − time))
```

(a) Check expiration for long-held pages

```
GLOBAL-HANDLER(MemoryZone zone)
 1  PageBlockList lowlist = NIL
 2  PageBlockList highlist = NIL
 3  for Integer id = 0 to MAX-ORDER(zone)
 4    BuddyOrder order = zone.freelist[id]
 5    ADJUST-REPLENISH-RATE(order)
 6    Integer blockdemand = BLOCKS-TO-TEST(order)
 7    if blockdemand ≠ 0
 8      PageBlock block = TAIL(order)
 9      while block ≠ HEAD(order) and blockdemand > 0
10        PageBlock prevblock = PREV(block, order)
11        if IS-TESTED(block) == FALSE
12          REMOVE(block, order)
13          if order ≤ LOW-ORDER
14            ENQUEUE(block, lowlist)
15          else
16            ENQUEUE(block, highlist)
17          N = N − 1
18        block = prevblock
19  SIGNAL-WORK(LOW-TEST-THREAD, lowlist)
20  SIGNAL-WORK(HIGH-TEST-THREAD, highlist)
21  SET-TIMER(this, replenish-δ)
```

(b) Test blocks for each buddy order

Fig. 3: Handlers for guarantee rate and global allocation

is incremented and *tested* is decremented when a block is allocated. The counters are set by the allocation timer handlers. Each counter is 16 bits which rarely saturates for small *replenish-δ* values.

C. Guarantee Timer and Handler

Figure 3(a) shows pseudo-code for the guarantee timer's handler. The handler checks whether a page is about to expire (i.e., $age(p)+\delta \geq test\_lifetime$). To check for test expiration, the handler compares the current time to a *timestamp* for each allocated physical page. When a page, p, is allocated, its *timestamp* is set to $t_{alloc}(p) + test\_lifetime - \delta$. On line 1, the current time, *time*, is queried. FIND-EXPIRED-PAGES returns a list of pages whose *timestamp* $\geq$ *time* on line 2.

Lines 3 to 8 migrate expired page by allocating a new tested page and copying the old one to it. During migration, page table entries (PTEs) for the old page are flagged. This action causes a process to fault when it touches a migrating page and to be paused until the migration completes. Once the old page is copied to the new one, the associated PTEs are updated and the old page is released.

Finally, line 9 re-arms the guarantee timer which is set to expire in δ time, taking page migration latency into account. As long as the handler finishes in δ, the promise on the vulnerability window is satisfied. However, our implementation does not guarantee this condition. δ is large enough (e.g., 10 seconds) that it is unlikely that the migration cannot be finished before the next timer event. If such a "deadline miss" actually arises, then migration can be parallelized for better performance.

D. Global Allocation Timer and Handler

To replenish the pool of tested pages for global allocation, COMeT regularly extracts and tests blocks of pages. Figure 3(b) shows pseudo-code for this process. GLOBAL-HANDLER is invoked with an identifier for the zone (*zone*). GLOBAL-HANDLER forms two lists: *lowlist* and *highlist* (lines 1 and 2) that collect untested blocks based on order rank. Blocks put into the lists are tested by the global tester (Section III-E). Lines 3 to 18 build the two lists. The **for** loop iterates over the buddy orders. A descriptor for an order is accessed on line 4.

Next, line 5 adjusts the replenishment rate ADJUST-REPLENISH-RATE (not shown) determines the new rate. If the rate is less than or equal the number of blocks requested, then the rate is adjusted upward to the actual demand. Otherwise, the current rate is decremented by 1. This approach scales the rate quickly upward, but avoids decreasing too rapidly, given the small time intervals over which demand is measured. Once the new rate is set, the number of blocks, *blockdemand*, to test to meet the rate is determined (line 6). BLOCKS-TO-TEST (not shown) computes this value as the difference between the current rate and the number of blocks that were tested in the last *replenish-δ* interval.

If blocks need to be tested (line 7), then up to *blockdemand* untested blocks are extracted from the order (lines 8 to 18). The **while** loop on line 9 iterates through the order's free list from tail to head. The list is processed in reverse because it is more likely that untested blocks will be at the list's end (these are the "older" blocks). An untested block is removed from the free list on line 12. If this block came from a low order (determined by the constant LOW-ORDER), then it is put into *lowlist* for testing. Otherwise, it is added to *highlist*. Lines 19 and 20 signal the low and high global tester threads that new blocks are available.

Finally, line 21 resets the global timer to *replenish-δ*. This parameter controls how frequently pages are pulled for testing; the page quantity (in blocks) is determined from the demand during the last *replenish-δ* interval. Unlike the guarantee rate, the replenishment rate does not have to be exact. Thus, the timer is simply set to expire in *replenish-δ* time.

E. Global Tester

GLOBAL-TESTER is the routine that tests page blocks. It waits until the global timer's handler produces work, and then it invokes TEST-BLOCK on each block in the work list to do a march test. To test a page, the page is first marked uncacheable to avoid the caches filtering accesses to physical memory. The march test operates on virtual addresses that are known to be mapped to the physical page. If the march test fails, then the block is split into two or more smaller blocks at the failed page. The new smaller blocks are assigned an appropriate number of pages (i.e., an integral power of 2), according to their order. The failed page is retired (never allocated again) by excluding it from the new blocks. If the test succeeds for all pages in a block, then the block is released to the allocator.

Baseline Configuration		Experimental Setup	
$test\_lifetime$	3*min* vulnerability window (2*min*, 4*min*)	Processor	Pentium 4 D (Presler 930)
δ	10*sec* (5*sec*, 20*sec*, 30*sec*)	Speed/cores	3.0 GHz, dual-core, no hyperthreading
$replenish$-δ	25*ms* (15*ms*, 35*ms*, 45*ms*, 0 for On-demand)	mem_cap, $page\_size$	1 GB, 4,096 bytes
α, mem_util, $test\_res$	1.0 (test all memory at maximum rate)	Linux kernel version	2.6.24.3
$test\_latency$	0.4*ms* (MATS march test was used [30])	Benchmarks	SPEC CPU2006, ref. input data sets
Minimum test rate	Same as guarantee rate		

TABLE I: Experimental setting

Program	Time	Util.	Program	Time	Util.	Program	Time	Util.	Program	Time	Util.
povray	751.5s	14.9%	hmmer	494.0s	24.7%	omnetpp	994.9s	35.2%	lbm	1340.6s	67.3%
caclulix	2557.2s	19.9%	soplex	570.9s	27.5%	gromacs	1632.7s	35.7%	milc	1242.0s	82.2%
namd	1173.0s	20.3%	perlbench	542.6s	29.0%	gamess	1632.0s	36.8%	mcf	948.3s	93.9%
gcc	111.6s	21.2%	sjeng	1634.1s	31.2%	libquantum	1789.6s	42.2%	gemsFDTD	1966.3s	95.4%
gobmk	193.7s	21.3%	astar	518.3s	31.3%	leslie3d	2179.2s	45.3%	bwaves	1405.2s	97.6%
h264ref	189.5s	22.5%	dealIII	942.7s	32.5%	xalancbmk	797.8s	52.6%	bzip2	261.8s	98.4%
sphnix3	1579.6s	24.1%	tonto	1782.8s	34.5%	zeusmp	1555.8s	62.8%	cactusADM	2395.7s	98.7%

TABLE II: Benchmark statistics (table is sorted by memory utilization)

IV. Evaluation

To determine COMeT's effectiveness, we investigated how many pages are tested, the performance overhead and energy for our implementation. We also studied several design parameters. We compared a naïve on-demand testing strategy (see Section II-B), COMeT and a baseline without testing.

A. Methodology

Table I lists on-demand testing and COMeT's base configuration. In the discussion of results, we refer to on-demand testing as "On-demand". We use a baseline 3 minute vulnerability window to evaluate On-demand and COMeT in a harsh situation where the test process is kept busy and system resources are taxed. In practice, the expected error rate is likely to be low enough that the vulnerability window can be set to a longer duration. We use the classic MATS march test [30]. For COMeT, a minimum replenishment rate is used to ensure some tested pages are available. For all lists, this rate is set to the guarantee rate. An initial replenishment rate for each free list is set on each program invocation. To get this initial rate, we profiled the benchmarks to find the minimum demand on each list. The minimum among all programs is used for a list's initial rate. To model On-demand, we fix the replenishment rate to 0. Page migration is enabled for On-demand.

The table also gives the experimental setup. Experiments are done in Linux single-user mode to minimize system activity. Because COMeT imposes overhead on system time, performance is measured as the sum of user and system time (i.e., wall-clock time). We also measure energy using a power meter. Energy is measured for a full SPEC run rather than individual benchmarks due to the meter's limited precision. Both cores of the experimental machine are used. The kernel can schedule the benchmark and test threads on either core to approximate a "core rich" environment. We used SPEC CPU2006 with reference inputs. Table II shows the benchmarks (sorted by memory utilization), their run-time ("Time"), and average memory residency ("Util.").

B. Overall Results

Pages Tested: Figure 4 shows the percentage of pages tested by On-demand and COMeT. We call this metric "page coverage". Page coverage is the ratio of the number of pages tested (free or used) to the total number of physical pages. On-demand's coverage is 15.3% to 99%, with an average of 46.4%. The coverage follows the trend of memory utilization. For example, *povray* has the smallest memory utilization (14.9%) and coverage (15.3%). Similarly, *cactusADM* has the highest utilization (98.7%) and coverage (99%). On-demand's coverage is slightly more than memory utilization since pages can be returned before expiration.

COMeT's coverage is 19.5% to 98.6% (51.4% average). The coverage follows the same trend as On-demand, tracking memory utilization. Again, *povray* has the lowest coverage and *cactusADM* has the highest. COMeT always keeps tested pages in the free lists, and naturally, its coverage is higher than On-demand. Coverage also indicates COMeT's prediction accuracy for future memory demand. The higher coverage relative to utilization reveals there is some overestimation from two aspects. First, COMeT scales up replenishment rate quickly but degrades it slowly. The delay in degrading the rate causes more pages to be tested than necessary. Second, COMeT conservatively keeps pages in rarely used free lists. It is desirable to overpredict by a small amount to have a reserve of tested pages for unexpected peak allocation.

Performance: Performance overhead for On-demand and COMeT is presented in Figure 5. This figure shows the slowdown experienced by a benchmark versus the baseline without testing. Slowdown is the ratio of a program's wall-clock time with testing to wall-clock time without testing.

On-demand's slowdown is 1.15 (*povray*) to 1.5 (*cactusADM*) with an average 1.34. The overhead generally increases with memory utilization. Because pages are sequentially tested with program execution, the full latency of allocating, testing, and mapping a page is observed, which harms execution time. *bzip2* has this behavior. It has a 1.48 slowdown due to its short execution time (261.8s) but high memory utilization (98.4%). There is less impact in *povray* because its resident size is only 14.9% of memory.

To enforce page lifetime, a large penalty is paid for migration in On-demand. When pages are migrated, target pages are tested on-demand, which dramatically increases migration latency. As a result, in programs with large working sets and high utilization, it is probable that a page under migration will be touched, forcing the program to be paused. In effect,

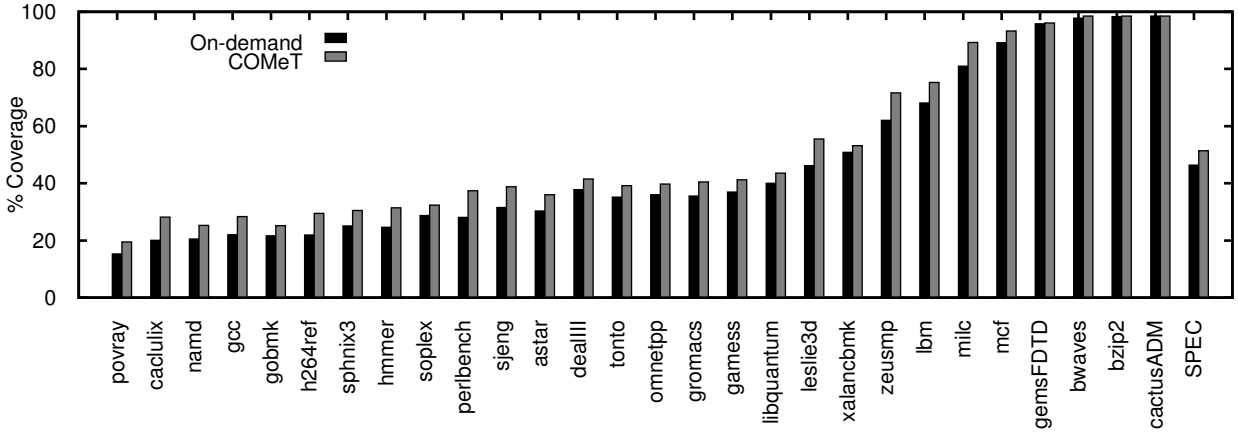


Fig. 4: Page coverage (percentage of physical pages tested)

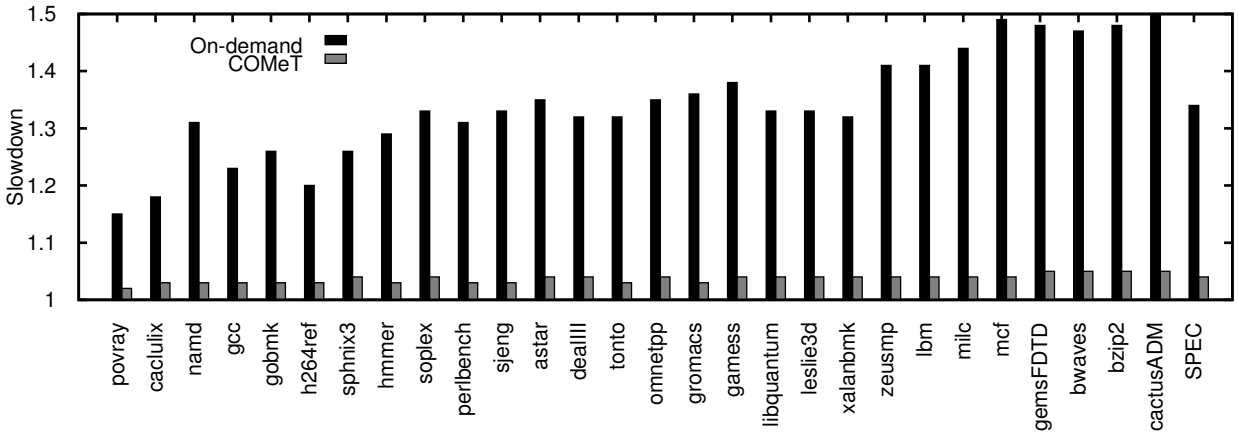


Fig. 5: Slowdown relative to baseline without testing.

the program cannot make quick progress since it executes sequentially with testing and migration. Several programs (e.g, *mcf*, *gemsFDTD* and *catcusADM*) exhibit this behavior.

Figure 5 also gives slowdown for COMeT, which is a modest 1.02 to 1.05, with an average of 1.04. Programs with high utilization again have the most overhead; however, the actual overhead is small. For example, *cactusADM* and *bzip2* use 98% of memory but suffer only a 1.05 slowdown. For *bzip2*, ahead-of-time testing has helped dramatically. When pages are requested, tested ones can be delivered without the latency of on-demand testing. Similarly, ahead-of-time COMeT has reduced migration latency because target pages are tested ahead of time. Migration is now primarily a page copy operation, which can happen at cache speed. This low latency leads to a smaller chance that a program has to be paused. Even if an application must be paused, it will be stopped for a much shorter duration than with On-demand.

COMeT's slowdown comes mostly from increased pressure on memory resources, including the hardware memory subsystem, despite the availability of a free core. This result demonstrates that ahead-of-time testing can efficiently test the experimental machine's 1 GB memory and maintain the pace of requests for tested pages.

Energy: We found that the total energy is 47% higher for On-demand than the baseline due to the extra time to run the programs. There is also some cost from more memory and core activity as shown by the 10% penalty in energy per hour with On-demand (98 vs. 108 watts). COMeT also increases total energy, but by a smaller 18%. COMeT's energy per hour (113 watts) compared to the baseline (98 watts) shows the impact of using a second core. In comparison to each other, COMeT pays more energy per hour than On-demand (113 watts vs. 108 watts) due to increased parallelism, but it has less slowdown, leading to better energy consumption. Note that the experimental machine is an early generation (Pentium 4) CMP, which is a challenging energy target. Current CMPs have less power dissipation per core, and thus, using an idle core for testing should have less energy overhead than these results suggest.

We conclude that ahead-of-time COMeT has feasibly low overhead. The remaining experiments focus on COMeT, and for brevity, a subset of SPEC with mixed behavior (*gromacs*, *zeusmp*, *mcf* and *calculix*) and a few other cases.

C. Configuration

COMeT has several important parameters, including vulnerability window duration, δ, and $replenish$-δ. We varied these parameters to observe their influence. The other parameters

were fixed according to Table I. We do not report coverage because it does not change from Figure 4.

Vulnerability window: The average "effective page hold time" is influential. This time reflects how long a program needs physical pages, including situations where a page is remapped frequently due to a high page fault rate and memory utilization. When a program has many active pages for a certain vulnerability window size, then it will have the same or more active pages at a smaller size, leading to more migrations. Thus, there is a greater likelihood that a program will need a page that is under migration, causing it to be briefly paused. For example, *calculix* is run with an input that causes many pages to be regularly allocated and released from the heap. It holds these pages for around 2 minutes, which causes its slowdown to increase from 1.03 (3 mins.) to 1.1 (2 mins.). *zeusmp* holds 100% of its pages for at least 3 minutes and goes from 1.04 (3 mins.) to 1.08 slowdown (2 mins.). *cactusADM* and *gemsFDTD* have higher page fault rates and utilization, which cause slowdown of 1.11 at 2 minutes. On average for all spec benchmarks, 2 minutes has the worst performance and 4 minutes has little gain over 3 minutes. Thus, we select a 3 minute window.

Guarantee timer (δ): For a 3 minute vulnerability window, we tried 5, 10, 20 and 30 seconds for δ. There was only a small overhead change as δ was varied (not graphed). For example, *zeusmp*'s slowdown was 1.04 at 5, 10 and 20 seconds. The overhead increased to 1.06 at 30 seconds. While the relative performance difference is small, varying δ does represent a trade-off. A small δ means that a page p will be migrated closer to its actual $t_{expire}(p)$ time. This can avoid migrations due to an unnecessarily early test expiration (for too large δ). However, a small δ causes more timer events, introducing extra overhead from the expiration check in GUARANTEE-HANDLER. δ also has to be large enough that expired pages can be migrated within the allotted δ time to invoke the guarantee handler. In SPEC, a 10 second δ best balanced the test expiration check's cost and early migration.

Replenishment timer ($replenish$-δ): The replenishment handlers (e.g., GLOBAL-HANDLER) are invoked periodically based on $replenish$-δ. Like δ, this parameter has a trade-off. A small value allows COMeT to react swiftly and it causes a *single* handler invocation to more quickly acquire a smaller number of pages than a large value. However, a small value may momentarily overreact to a spike in demand. It also leads to more handler invocations. Thus, there can be more *total* competition for memory resources.

To investigate this trade-off, we tried 15*ms*, 25*ms*, 35*ms*, and 45*ms* for $replenish$-δ. All programs had little difference for 25*ms*, 35*ms* and 45*ms*, but a larger change was observed for 15*ms*. At 25*ms*, slowdown is 1.03 to 1.05, with a 1.04 average. Slowdown is higher for 15*ms*: 1.05 to 1.11 with a 1.06 average. Programs with high memory utilization or page turn-over (e.g., *calculix*, *gemsFDTD* and *bzip2*) are the most harmed at 15*ms*. It is better to pull more pages less frequently (at 25*ms*), but this effect diminishes as utilization decreases. *gromacs* and *zeusmp* have moderate utilization, with many long-held pages, and suffer less penalty from 15*ms*. Interestingly, in these programs, less replenishment is traded for more migration. Even in the lowest utilization cases (e.g., *povray*), there is still some penalty for a 15*ms* interval from extra handler invocations. From these results, we find that a 25*ms* interval is the best choice since it lets COMeT respond more quickly to memory demand than 35*ms* and 45*ms* but has less slowdown than 15*ms*.

D. Test Guarantee and Replenishment

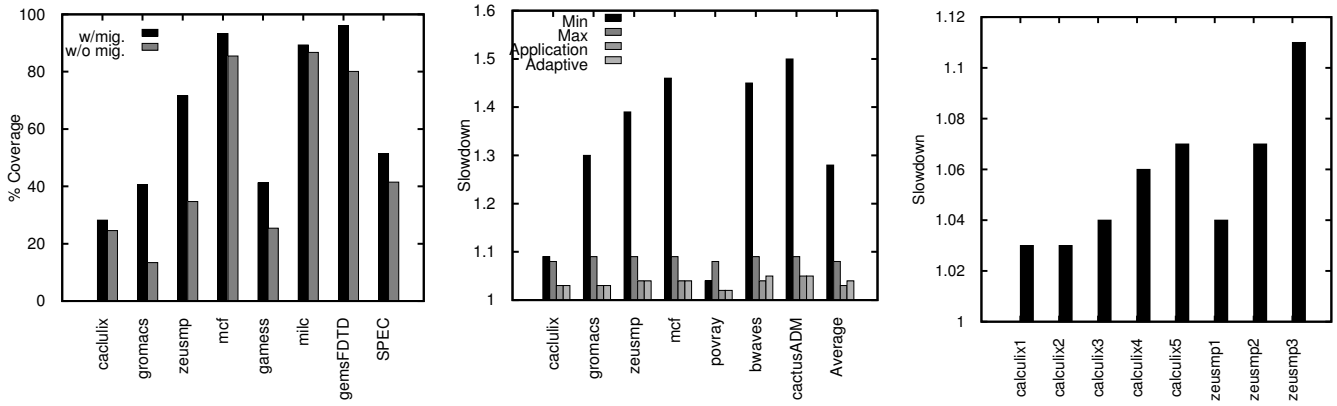
Page migration and replenishment are key aspects to COMeT: its ability to limit exposure to errors partly depends on migration to test long-held pages and its overhead depends on adapting the replenishment rate.

Page migration: Figure 6(a) shows page coverage without and with page migration. Some programs have a large reduction when page migration is disabled (e.g., *zeusmp*'s coverage goes from 71.6% to 34.7%). These programs hold many pages beyond the vulnerability window, and thus, migration is especially valuable. *calculix*, *mcf* and *milc* have different behavior – migration has a smaller impact due to page turnover (which leads to a small number of long-held pages). *gemsFDTD* has high memory utilization with a small number (7.3%) of long-held pages, and thus, its coverage is moderately reduced (96.1% to 80.1%). The average coverage reduces from 51.4% to 41.5%. From these results, we conclude that page migration is necessary to limit error exposure.

Replenishment rate: Figure 6(b) illustrates the benefit from an adaptive replenishment rate. The figure compares slowdown for four rate policies: "Min" statically fixes each free list's rate to the minimum average request rate among all applications, "Max" fixes the rates to the maximum averages among all applications, "Application" fixes the rates to the average rates for a given program, and "Adaptive" is COMeT's scheme. "Min" does well for low demand programs; e.g., *povray* (1.04 slowdown). However, because there is a drought of tested free pages, most programs (e.g., *cactusADM*) suffer badly. Pages are frequently tested on demand, causing "Min" to behave similar to D-COT. "Max" corrects this problem; the rates are set high enough that the free lists are replenished. Nevertheless, "Max" does worse than "Application" due to pressure from unnecessary testing. "Application" is the best because the replenishment rates are set specifically for each program. "Adaptive" comes close to "Application", but it does not need offline profiling. The SPEC average is 1.03 for "Application" and 1.04 for "Adaptive". The adaptive scheme can dynamically change to global demand, walking the line between too little and too much testing. For this reason, we believe that an adaptive replenishment rate should be used.

E. Overload Behavior

To examine COMeT in an overload situation, we measured performance as system load is increased by executing multiple program instances. *zeusmp* and *calculix* are used because they have enough memory pressure to create stress without exhausting most physical memory in one instance. Figure 6(c)



(a) Page coverage due to migration (b) Effect of replenishment rates (c) Slowdown under overload

Fig. 6: Effect of Migration, Replenishment Rates and Slowdown

shows the results. Slowdown is computed relative to the same number of instances without testing. The program instances can execute on both processor cores.

The figure shows that slowdown increases with load. With a single *zeusmp* instance, the slowdown is 1.04, which increases to 1.11 with three instances. The slowdown for *calculix* goes from 1.03 to 1.07. As more instances are introduced, allocation rate and memory utilization increase, causing competition for the cores and memory subsystem. Core competition is the dominate factor; the cycle cost of testing is observed at more instances because the cores could otherwise be used to execute the programs without testing. Although there is an increase, the maximum slowdown is only 1.11. We find that COMeT behaves well in overload.

F. Detecting Errors

To evaluate COMeT's ability to detect errors, we performed experiments that inject an increasing number of emulated errors in memory. The goal is to determine how much longer a program will run with COMeT. We model an aggressive scenario with a high number of errors to stress COMeT.

We injected 200 to 1,600 errors in steps of 200 with a framework for memory error emulation [18]. From a profile, we determined the physical pages that were likely to be used, and these pages received the errors. Errors were selected using a random distribution and inserted at word granularity. The same error addresses were used in execution runs with and without COMeT. Because the memory allocator may map virtual pages differently on each run, we used Monte Carlo trials. For each error amount, 20 trials were done with and without COMeT. In each trial, program execution was "warmed up" for 2 minutes. After warm-up, all errors were inserted (on free and used pages). The program was then executed for 10 more minutes. We limited execution due to long run-times for so many trials. We used (TTFE) for *mcf* and *bwaves* to get "time to first error". These programs were chosen based on their high memory utilization.

Without COMeT, neither benchmark reached 10 minutes without an error. At 200 injected errors, *mcf* had 488 TTFE and *bwaves* had 533 TTFE. The difference in execution times is related to memory access frequency and number of unique addresses touched. When the number of injected errors is increased, the programs access an error location earlier. At 1,600 errors, the TTFEs for *mcf* and *bwaves* are 129 and 155.

When COMeT is enabled, the programs are more likely to reach the 10 minute point without an error—at 800 or less emulated errors, both programs executed the full 10 minutes. At 1,000 errors, *bwaves* encounters its first error, with 559 low and 596 average TTFE. *mcf* hits its first bad address at 1,200 injected errors (520 low and 590 average TTFE). At high counts, it is probable that an error is inserted on an actively used page, which leads to a TTFE below 600. Other benchmarks had a similar trend.

This experiment demonstrates COMeT's benefit and limitation. When the error count is high, an error will likely appear on an in-use page during the vulnerability window. If the page is heavily used, with all locations referenced, then a corruption may happen. This behavior happens at 1,000 or more errors for *bwaves* and *mcf*. Note that the experiment's error counts are actually extreme—errors happen *all at once throughout memory*. In reality, errors would probably manifest more gradually and COMeT would likely detect them, as the lower error count results suggest. COMeT, like other diagnostics schemes, does not guarantee an error-free condition. Instead, it reduces the chance of encountering an error. COMeT let the programs execute 1.13-3.4x (*bwaves*) and 1.23-4.41x (*mcf*) longer than no testing, and we conclude that COMeT does increase protection against memory errors.

V. Related Work

It has been predicted [3, 4] and empirically observed [5, 23] that processor and memory chips built at nanometer-scale can suffer frequent hardware errors. Both smaller transistor geometry and higher chip density aggravate the problem. A large field study done by Schroeder et al. [23] revealed that DRAM error rates are surprisingly high, with 25,000 to 70,000 errors per billion device hours per Mbit and more than 8% of memory modules affected by errors per year. They provide strong evidence that memory errors are dominated by hard errors. Other field studies [12, 27] make similar observations.

While these studies strongly motivate our work, they do not propose a specific technique to combat errors in DRAM chips.

Reliability in the processor has been well studied [9–11, 13, 17, 19, 22, 25] but there has been less work for online memory testing. For main memory, the most viable hardware approach relies on embedding information redundancy in DRAM. Such redundancy provides a check on data integrity and/or a capability to restore the original data on detecting an error(s). For example, the most popular DRAM protection practice employs error correction codes (ECC). A typical ECC is capable of correcting a single-bit error and detecting two errors in a single error correction unit (typically 64 bits), commonly referred to as SECDED. Other schemes [7, 32] distributes memory content (and redundancy) across different chips (or different DIMMs) to provide stronger protection against multiple bit errors. For example, IBM's chipkill can scatter memory bits in a single ECC unit to multiple chips. A smart "virtualized" embodiment of the ECC scheme [31] was recently proposed to decouple the actual ECC from the data in memory and separate the process of detecting errors from the rare need to also correct errors (to save energy).

Two previous schemes are directly related to our work, Elm et al. [8] and Singh et al. [24]; like our work, they propose a software memory test strategy. While both proposals and COMeT implement a memory tester in the OS, the goals and strategies are different. First, the previous proposals assume ECC and test memory occasionally to catch faults with no consideration for page migration. We do not assume ECC and test memory continuously to mask errors that occur dynamically. Second, both previous proposals are not adaptive; they allocate a fixed chunk of memory for testing at a fixed rate the system administrator assigns. They do not relate test strategy parameters and error coverage. Third, they do not study memory test performance on individual programs. For example, Elm et al. measured and reported "system performance degradation" (essentially, lost cycles) due to their memory tester during a one-week experiment with 25 complete tests of a 32MB main memory. Singh et al. employed two programs but detailed memory access behaviors were not discussed or related with the coverage results. Lastly, these previous studies did not consider CMPs.

VI. Conclusion

This paper presents a transparent software strategy, COMeT, for diagnostic memory testing. To achieve low run-time overhead, COMeT uses an available core in a CMP to proactively test memory ahead of its allocation. The scheme also uses concurrent page migration to limit the exposure of in-use pages to errors. An implementation of the approach is described and evaluated. When emulated errors were injected, we found that applications executed 1.13 to 4.41 times longer with COMeT than without it. This improvement in resilience comes with a minimal average slowdown of just 1.04 for SPEC CPU2006. The results show that a software-only approach to check memory health can be incorporated in a transparent fashion to have feasilbly low performance impact.

References

[1] Memtest86+. http://www.memtest.org/.

[2] M. Abbott et al. Durable memory RS/6000 system design. In *Int'l. Symp. on Fault-Tolerant Computing*, 1994.

[3] S. Borkar. Microarch. and design challenges for gigascale integration. In *MICRO*, 2004.

[4] S. Borkar et al. Parameter variations and impact on circuits and microarchitecture. In *MICRO*, 2003.

[5] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *MICRO*, 23(4):14–19, 2003.

[6] K. Constantinides et al. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation. In *MICRO*, 2007.

[7] T. J. Dell. A white paper on the benefits of chipkill. In *IBM Microelectronics Division*, 1997.

[8] C. Elm et al. Automatic on-line memory tests in workstations. In *MTDT Workshop*, 1994.

[9] S. Gupta et al. Adaptive online testing for efficient hard fault detection. In *ICCD*, 2009.

[10] M.-L. Li et al. Understanding the propagation of hard errors to software and implications for resilient system design. In *ASPLOS*, 2008.

[11] X. Li et al. Online estimation of architectural vulnerability factor for soft errors. In *ISCA*, 2008.

[12] X. Li et al. A realistic evaluation of memory hardware errors and software system susceptibility. In *USENIX Conf.*, 2010.

[13] Y. Li, O. Mutlu, and S. Mitra. Operating system scheduling for efficient online self-test in robust systems. In *ICCAD*, 2009.

[14] A. Messer et al. Susceptibility of commodity systems and software to memory soft errors. *IEEE Trans. on Computing*, 53(12):1557–1568, Dec 2004.

[15] S. S. Mukherjee et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO*, 2003.

[16] S. S. Mukherjee et al. Cache scrubbing in microprocessors: Myth or necessity? In *PRDC*, pages 37–42. IEEE Computer Society, 2004.

[17] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA*, 2002.

[18] M. Rahman, B. Childers, and S. Cho. StealthWorks: Emulating memory errors. In *Int'l. Conf. on Runtime Verification*, 2010.

[19] S. Reinhardt and S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA*, 2000.

[20] P. Reviriego et al. Optimizing scrubbing sequences for advanced computer memories. *IEEE Trans. on Device and Materials Reliability*, 10(2):192–200, June 2010.

[21] A. M. Saleh et al. Reliability of scrubbing recovery-techniques for memory systems. *IEEE Trans. on Reliability*, 39(1):114–122, April 1990.

[22] S. Sastry Hari et al. mSWAT: low-cost hardware fault detection and diagnosis for multicore systems. In *MICRO*, 2009.

[23] B. Schroeder et al. DRAM errors in the wild: A large-scale field study. In *Int'l. Conf. on Measurement and Modeling of Computer Syst.*, 2009.

[24] A. Singh et al. Software based in-system memory test for highly available systems. In *MTDT Workshop*, 2005.

[25] D. Sorin et al. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA*, 2002.

[26] J. Srinivasan et al. The case for lifetime reliability-aware microprocessors. In *ISCA*, 2004.

[27] D. Tang et al. Assessment of the effect of memory page retirement on system ras against hardware faults. In *DSN*, 2006.

[28] A. van de Goor and I. Tlili. March tests for word-oriented memories. In *DATE*, 1998.

[29] R. van Rein. BadRAM: Linux kernel support for broken RAM modules, site last visited July 11, 2010. http://rick.vanrein.org/linux/badram/.

[30] C.-F. Wu et al. Simulation-based test algorithm generation for random access memories. In *IEEE VLSI Test Symp.*, 2000.

[31] D. H. Yoon and M. Erez. Virtualized and flexible ECC for main memory. In *ASPLOS*, 2010.

[32] H. Zheng et al. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. In *MICRO*, 2008.