Efficient Online Memory Error Assessment and Circumvention for Linux with RAMpage

Horst Schirmeier*, Ingo Korb, Olaf Spinczyk and Michael Engel

Department of Computer Science 12,
Technische Universität Dortmund,
Otto-Hahn-Str. 16, 44221 Dortmund, Germany
E-mail: horst.schirmeier@tu-dortmund.de
E-mail: ingo.korb@tu-dortmund.de
E-mail: olaf.spinczyk@tu-dortmund.de
E-mail: michael.engel@tu-dortmund.de
*Corresponding author

Abstract:
Memory errors are a major source of reliability problems in computer systems. Undetected errors may result in program termination or, even worse, silent data corruption. Recent studies have shown that the frequency of permanent memory errors is an order of magnitude higher than previously assumed and regularly affects everyday operation.

To reduce the impact of memory errors, we designed RAMpage, a purely software-based infrastructure to assess and circumvent permanent memory errors in a running commodity x86-64 Linux-based system. We briefly describe the design and implementation of RAMpage and present new results from an extensive qualitative and quantitative evaluation. These results show the efficiency of our approach – RAMpage is able to provide a smooth graceful degradation in the presence of permanent memory errors while requiring only a small overhead in terms of CPU time, energy, and memory space.

Keywords: Memory errors; Software-based fault tolerance; DRAM chips; Silent data corruption; Operating systems; Reliable operation

Biographical notes:

Horst Schirmeier received his Diploma in Computer Science from Friedrich-Alexander-Universität Erlangen, Germany. He worked as a researcher in the System Software group at FAU Erlangen, and is currently working in the field of resilient infrastructure software and fault resilience assessment for the *DanceOS* project in the Embedded System Software group at Technische Universität Dortmund, Germany.

Ingo Korb received his Diploma in Computer Science from Technische Universität Dortmund. He is a researcher in the EU MADNESS project at the Informatik Centrum Dortmund (ICD). His research interests include software-based fault tolerance and real-time CPU emulation.

Olaf Spinczyk received his PhD in Computer Science from Otto-von-Guericke-Universität, Magdeburg, Germany. He worked as a senior researcher in the Operating Systems and Distributed Systems group at Friedrich-Alexander-Universität Erlangen, Germany, and is now full professor and head of the Embedded System Software group at

Technische Universität Dortmund, Germany. His research is focussed on the construction principles as well as programming language and system software support for highly efficient and reliable embedded systems.

Michael Engel received his PhD in Computer Science from University of Marburg, Germany. He is currently a senior researcher at the Design Automation for Embedded Systems group at Technische Universität Dortmund, Germany, where he leads two research projects in the context of dependable embedded systems. His research interests include dependability for embedded systems, energy efficiency, embedded hard- and software design and operating systems.

1 Introduction

Memory reliability has been a major concern for the reliable operation of computers for the last few decades. However, for a long time, credible values on error rates have been hard to obtain and were mostly derived from small-scale experiments. A recent large-scale study using Google's server farm by Schroeder et al. (2009) came to the conclusion that permanent errors in off-the-shelf memory DIMMs (Dual In-Line Memory Modules) are an order of magnitude more common than previously assumed. The study concluded that on average, a permanent error is expected to show up once a year in a standard 1 GiB DDR memory module. Thus, a single-bit error is expected to show up once every few months in a system using multiple memory modules.

In the best possible case, such an error causes a program or operating system crash or malfunction, which is comparatively easy to detect. However, in an analysis of errors in CERN's computer systems, Panzer-Steindel (2007) showed that the probability of *silent data corruption* errors – i.e., undetected errors that result in unexpected value changes – is several orders of magnitude higher than expected by component failure statistics.

This observation is also supported by a recent study by Nightingale et al. (2011), which states that DRAM errors in consumer-grade systems are far more likely than expected from an analysis of radiation effects. They confirm the validity of the bathtub curve assumption for DRAM components, i.e., the failure rate of RAM chips increases sharply beginning at a certain age of the components.

While high-end systems usually employ ECC or other hardware-based error correction methods, these are prohibitively expensive for small-scale, cost-sensitive systems. However, reducing the probability of undetected permanent memory errors is also a worthwhile goal for these systems. Systematic memory tests are a useful tool to detect errors; if these tests are performed with a sufficiently high frequency, we expect most errors to be discoverable. However, shutting down a system in order to perform tests, e.g., using software like Memtest86+ (Demeulemeester (2011)), is in most cases unacceptable.

As a consequence, we developed RAMpage, an approach to run a memory tester as a system service *during normal operation.* Compared to existing memory test mechanisms, RAMpage requires no downtime and allows for flexible selection of memory test frequency and test methods, and works mostly as a user-space process.

This article is an updated and expanded version of our paper "RAMpage: Graceful Degradation Management for Memory Errors in Commodity Linux Servers" (Schirmeier et al. (2011)). The new material includes a substantially more effective port of RAMpage to Linux 3.5, a completely revised and expanded evaluation section, including systematic fault-injection runs proving the validity of our approach, an analysis of RAMpage's impact on energy consumption, and more benchmark results, as well as an updated overview of related work.

The outline of this paper is as follows. Section 2 describes memory testing methods and system support for graceful degradation in Linux. An overview of the structure and implementation of RAMpage is given in sections 3 and 4. The effectiveness and efficiency of in-system memory tests are evaluated in section 5. Section 6 discusses related work; finally, section 7 concludes the paper and gives an outlook to ideas for future research.

2 Graceful Memory Degradation in a Linux Environment

Graceful degradation management provides a best-effort approach to operate a system as long as possible, albeit potentially with reduced functionality or performance. RAMpage performs memory tests at runtime and implements graceful degradation by marking memory areas containing defects as "poisoned", which guarantees that neither the operating system nor application processes will continue to use them. In the following subsections, we give a short overview of approaches to memory testing and of Linux-specific support for degradation management.

2.1 Memory Testing

Memory testing is a well-explored research area. Examples of test algorithms can be found in early papers by Nair et al. (1978), Hayes (1975) and Srini (1978), as well as in more recent publications (Chang et al. (1989), Yoon & Erez (2010)). Based on models for static memory faults, algorithms of increasing complexity can find errors ranging from simple stuck-at faults that permanently tie a bit to a value up to complex, correlation- and timing-based faults.

Most of the memory test algorithms described in the literature require destructive write access to the memory range that is examined. Thus, in contrast to ECC scrubbing approaches (see subsection 6.1) that only read memory cells and check for ECC errors, a universal memory testing infrastructure has to possess knowledge about the current usage state of the memory region to be tested. Obviously, destructive write tests can only be performed if the memory to be written into is not in use by the kernel or applications. RAMpage thus has to ensure the memory areas to be tested are free before testing can commence.

However, modern Unix-like systems tend to use almost all physical memory for the (unified) buffer caches and VM cache. A memory tester thus has to "liberate" page frames from the system – either from the kernel or the running processes. In theory, this is possible for all page frames, since kernel and application memory accesses occur using logical memory addresses. In practice, however, it is not always possible to gain access to all available memory in a system. For example, DMA buffers for I/O devices, memory with MMU page tables, and other special kernel allocated page frames are usually not relocatable (see Schirmeier et al. (2011) for possible solutions in future systems).

Another question influencing the design of RAMpage is how many pages need to be tested at the same time, e.g., to find correlation errors. Since tested page frames are not available to the rest of the system during the test, the policy is to only allocate as many pages as required and release pages when the test is finished.

2.2 Linux Support

Virtual memory allows the operating system to control physical memory use on a fine granularity, i.e., the size of a page frame, typically a few kilobytes, depending on the architecture. Linux implements several competing – and partially redundant – frameworks that can handle hardware errors and changing system configurations. We describe the various existing approaches below and evaluate their advantages and shortcomings for use in our memory testing system.

EDAC

Support for reacting to faults in the Linux kernel was originally derived from separate modules handling memory ECC and PCI-bus parity errors. The so-called EDAC (Error Detection and Correction) framework now encompasses this functionality in recent kernel versions. However, its functionality is so far quite restricted. In the case of ECC errors, the error can be logged and the affected memory cell re-tested; in case of PCI parity errors, the affected I/O transfer can be repeated. Currently, this framework does not provide general interfaces to test and possibly disable other components of a system. While this is envisioned for future releases, Linux EDAC support is at the moment not useful as a basis for memory degradation management.

Memory hotplugging

Memory hotplugging enables the addition and removal of physical memory at runtime. This is useful for special systems that in fact allow hotplugging and -removal of memory modules as well as for virtual machines with changing memory requirements.

Hotplugging may also be employed to remove bad memory regions from kernel use and future allocation. However, since hotplug's remove operation was intended to remove complete physical memory modules from a system, the achievable granularity is rather coarse. In the current implementation, only regions of 1024 contiguous pages (4 MiB on an x86 machine with 4 kiB page size) can be removed together. It turns out, though, that parts of the hotplugging infrastructure come in quite handy for "liberating" page frames for testing that are currently in use, as we describe in section 4.

hwpoison

The Linux *hwpoison* framework was originally intended to support recovery from memory errors signaled by recent Intel CPUs. Using this framework, a memory page frame can be declared "poisoned", which results in killing any process associated with that page, and precludes its further allocation. While the framework is intended to be used with ECC-enabled memory controllers, the available functions to taint single page frames make it possible to remove selected 4 kiB regions (on x86-architecture machines) containing faulty memory cells. Furthermore, *hwpoison* provides an additional method for claiming allocated page frames, which makes it even more suitable for our memory tester.

3 System Structure

Memory testing systems can be implemented in various different ways – e.g., there exist pure kernel-mode as well as pure user-mode approaches (see section 6 for a detailed discussion). In the following subsections, we describe the requirements to our memory testing system as well as problems restraining some design options, and detail the resulting infrastructure components and their interaction.

3.1 Design Considerations

RAMpage's central design idea was to perform online memory testing for current commodity machines that requires as few changes to the operating system as possible. Since many server systems run the Linux operating system, basing our infrastructure on this platform was an obvious choice.

RAMpage is designed to require only a minimal set of functionality in kernel mode, giving a maximum flexibility to adapt the system's components running in user mode. Thus, only critical, system-level functionality – such as "liberating" (find details on how this is done in section 4) and allocating physical memory, and disabling the CPU cache for the page frames to be tested – is implemented as a dynamically loadable kernel module. Most of the functionality, especially the tests themselves, executes in user mode.

The RAMpage user-space component does not perform memory tests itself, but delegates them to a memory test plugin. This flexibly exchangeable plugin determines the physical frame numbers to be tested and performs the specific tests. The user-space component itself takes care of common operations: requesting and mapping page frames, returning good memory to the kernel and marking bad memory. Currently, RAMpage provides ports of all tests available in Memtest86+ (cf. fig. 3).

Implementing the major part of the tester as a user space component has an additional advantage. Dealing with a normal user-mode process, a system administrator can directly control the performance impact by using standard Unix methods (such as `nice`).

However, several architectural properties as well as Linux idiosyncrasies also influenced the design. The impact of these is described in the following subsection.

3.2 Obstacles Influencing the Design

Performing memory tests that operate on a page-frame basis requires safe access to physical memory, which is usually not supported by current operating systems like Linux. While tests completely executing in user mode would be preferable – no

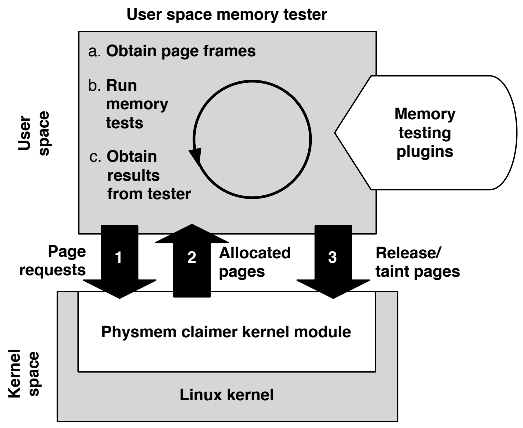

Figure 1 RAMpage's overall structure: The user-space memory tester with exchangable testing plugins consults the *physmem* kernel module to obtain physical page frames, and to taint frames positively tested for defects.

kernel modifications are required, reducing critical in-kernel code to zero – *safely* running such tests while providing access to physical memory cannot be guaranteed.

The page frames under test, thus, are required to be unused by the kernel or other processes. This can easily be guaranteed by allocating the memory pages to the testing process itself. However, requesting memory from user space is only possible using system calls such as `mmap`, which provide a set of *virtual* memory pages. The kernel does not provide information on the physical location of these virtual memory pages to the user mode process; in fact, the Linux VM system does not even guarantee that the logical address space is backed by physical memory at every point in time. While a user space test could try to reserve as many pages as are available, it cannot be guaranteed that this method actually covers a large amount of physical page frames. In addition, the memory reserved for testing would not be available to other applications, which may result in reduced system performance. Therefore, a memory tester running in user space alone may only actually test a small percentage of the total *physical* memory.

In consequence, a memory tester has to be able to request *physical* page frames in order not to repeatedly re-test only a small memory area. Knowledge of physical addresses is also required for providing hints to the graceful degradation handling mechanism. RAMpage therefore implements a kernel module to allocate physical page frames to the user mode testing component.

3.3 RAMpage Components

RAMpage's overall structure is shown in fig. 1. The system consists of two major components, the *physmem* kernel module that handles page frame allocation, deallocation and tainting, and the user-mode component that performs the actual memory tests running as a normal process among other user-mode processes.

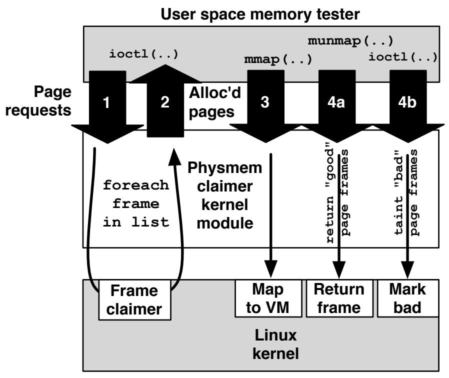

Figure 2 RAMpage's *physmem* page frame claimer kernel module and its system call interface.

3.3.1 Kernel–User Space Interaction

The tests run in a configurable loop in the userspace tester. Before each test pass, it requests a set of physical page frames to be tested by the specific memory-test plugin (a).

This list of page frame numbers is passed to the *physmem* kernel module (1), which tries to obtain the requested page frames from the frame claimer (see section 4 for details on the different claiming methods) and returns a list of successfully claimed page frames to the tester (2). The tester then maps the successfully requested page frames into its virtual address space using a standard `mmap` system call. The tester passes the page frame list to the memory testing plugin that implements the specific test algorithm (b).

After testing is finished, the tester obtains a result list from the testing plugin that indicates the test results: successfully tested vs. error detected (c). The successfully tested page frames are then deallocated using the `munmap` system call, whereas page frames that contain a detected error are reported to the kernel module (3). The module then marks these page frames as "bad" to avoid further allocation.

The structure of the kernel module is shown in fig. 2. The module uses a device driver interface which implements the `ioctl` and `mmap` system calls for the device (`/dev/physmem`). The module waits for commands passed by the user space via one of two `ioctl` calls to request a set of physical pages (1, 2) or to taint a page frame (4b). These commands are carried out via the frame claimer respectively the *hwpoison* subsystem. Mapping the page frames to be tested into the user space tester's virtual memory is done by a standard `mmap` system call (3) performed on the device; successfully tested pages are simply unmapped from the tester's memory space using the `munmap` system call implemented by the kernel module (4a).

3.3.2 Scheduling of User-Level Tests

Providing a plugin interface for different memory tests provides an important degree of freedom for a system administrator who wants to run online memory tests. However, the scheduling of tests is as important as the test algorithm selection,

Test #	Description
0	Address test, walking ones *(ineffective)*
1	Address test, own address
2	Moving inversions, ones & zeros
3	Moving inversions, 8-bit pattern
4	Moving inversions, random pattern
5	Block move, 80 moves *(ineffective)*
6	Moving inversions, 32-bit pattern
7	Random number sequence
8	Modulo 20, random pattern

Figure 3 Memtest86+ tests ported to RAMpage. Tests #0 and #5 cannot be run effectively as they need access to a large fraction of physical memory at the same time.

since this has the potential to heavily influence performance characteristics of the tester.

RAMpage provides several different schedulers, most notably the “blockwise” and the “slow” scheduler. The “blockwise” scheduler batches blocks of 512 page frames together and tests these in a burst. The test is only performed if the last successful test lies a specified time in the past; for benchmarking purposes this scheduler can also be set to “full-speed”, resulting in continuous memory tests. In contrast, the “slow” scheduler is intended to significantly reduce the memory testing overhead. This scheduler tests blocks of 4096 page frames (16 MiB) each and distributes tests of different blocks so that the complete physical memory of a system is tested within 24 hours (or any other configurable timeframe).

4 Implementation Details

RAMpage is implemented in ANSI-C for the kernel module and the performance-critical test algorithms, and Python for the user-space testing infrastructure. The kernel module contains 1,285 lines of C code, whereas the user-space is implemented in 1,985 lines of Python code. The memory tester plugins, a port of the well-established Memtest86+ (Demeulemeester (2011)) tests (fig. 3), add up to 909 lines of C code. One limitation is that the address line test #0 and test #5 cannot be run effectively in our infrastructure, since they require access to a large fraction of physical memory at the same time. This would defeat our design goal of minimal-impact online testing, and is not really a severe drawback: If a system’s memory address lines are malfunctioning, a complete system failure is inevitable in most cases anyways.

In the following subsections we elaborate on some of the roadblocks that showed up during the implementation, and explain Linux kernel details where necessary.

4.1 Memory Management

As stated earlier, actually getting access to unused and especially used memory page frames is one of the harder problems to be solved when implementing a memory tester. Since almost all physical memory is allocated to the kernel, processes, or the buffer cache, there is only a small amount of actual free memory available at runtime. Our memory tester thus has to claim memory from the kernel in order to perform tests, and return the allocated frames after testing.

Linux manages allocation of memory through the page allocator, which splits the physical memory into zones. Each of the zones is maintained by a *buddy* allocator, which manages memory in blocks sized in powers of two. In Linux, the base unit for the buddy allocator is a page frame, so the managed objects have sizes of 4 kiB, 8 kiB, 16 kiB, etc.

The page allocator is used to allocate MMU page table entries for virtual memory management. All other memory allocators in Linux (such as SLAB, described by Bonwick (1994)) rely on the page allocator to claim memory. Its most important user is the page cache, which contains caches of file data as well as anonymous memory (i.e., memory that is not related to any file) used by processes, including all text and data segments, read and written file contents, and meta data.

Important memory areas that are *not* managed by the page allocator include kernel text and data, as well as memory used during the boot process. These are managed by the separate *bootmem* allocator.

4.2 Claiming Page Frames

RAMpage's user-space component obtains a set of physical page frame numbers from the currently employed test scheduler. This set is passed to the kernel memory claimer module, which tries to allocate the frames using the kernel's buddy allocator. When they have successfully been allocated by the kernel module, it marks them as non-cacheable – otherwise the memory tester would mostly test the consistency of the CPU's data caches. The corresponding list of frames is returned to the user-space component, which then can map the successfully claimed page frames into its virtual address space and commence testing.

It turns out that the buddy claimer alone fluctuates strongly in whether it succeeds to claim page frames or not, depending on how much memory is currently being used, and for what purpose (cf. section 5). Therefore we complemented the claimer with two additional claiming methods, both more intrusive and side-effect–prone but also more effective in some situations: 1. Linux's memory hotplugging infrastructure provides an implementation for offlining large chunks of memory (multiples of 4 MiB) by migrating its contents to other areas (originally intended for physical removal of RAM modules at runtime). 2. The *hwpoison* subsystem contains a page-“shaking” function for (likewise *non*-destructively) liberating a single page frame for subsequent claiming via buddy or hotplugging. (Note that this excludes the destructive part of *hwpoison* that resorts to killing processes when claiming fails, cf. section 2). Due to their potential for harming system performance (e.g., the hotplug claimer contains a code path that completely drops the better part of the buffer cache), these additional measures can be configured by the user-space tool for each page frame to be claimed.

4.3 Page Frame Testing and Poisoning

The test of a page frame can have one of two outcomes. In most cases, the test runs without detecting any error, so it can be returned to the kernel for further use. This

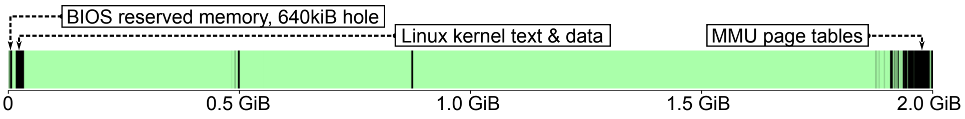

Figure 4 Single-bit stuck-at memory errors detected by RAMpage in fault-injection experiments conducted with Fail*: Only 5.3% of the faults injected at every second 4 kiB page boundary were not located (black areas) due to unsuccessful page claiming.

happens by simply unmapping the page, which is then automatically repossessed by the kernel's memory management.

In case the memory test detects one or more errors in a page frame, however, degradation management is being initiated. Instead of returning the affected page frame to the kernel for reuse, the page has to be specially marked to ensure it will not be allocated again until system reboot. Marking memory as "bad" is performed by the kernel module utilizing the *hwpoison* framework.

5 Evaluation

We evaluated RAMpage under several qualitative and quantitative aspects. The tester's effectiveness was evaluated by simulating defects in a virtual machine as well as using real defective RAM modules. RAMpage's efficiency was assessed by measuring both maximum test throughput and physical memory coverage for different claiming methods in varying system load scenarios. Finally, a practicality analysis was conducted by measuring performance, latency and energy consumption side-effects on a set of standard benchmarks.

Unless mentioned otherwise, all measurements were conducted on a standard Dell Optiplex 755 PC (Intel Core 2 Quad Q9550 CPU at 2.83 GHz, 2 GiB DDR-2 800 RAM) running Debian Linux 6.0 with a 64-bit x86-64 SMP-enabled Linux 3.5 kernel (slightly modified to export some internal symbols needed by our module, and to timeout faster when trying to hotplug memory). The following subsections describe the experiments we conducted and the conclusions we draw.

5.1 Effectiveness: Virtually and Really Broken Hardware

In order to assess RAMpage's effectiveness, we conducted a series of automated tests with Fail* (Schirmeier et al. (2012)), our fault-injection experiment framework, configured with QEMU v1.1 as the x86-64 simulator backend. We implemented an experiment campaign running one experiment for every second 4 kiB page boundary in a 2,047 MiB address space, injecting a stuck-at-1 single-bit fault and observing whether RAMpage (with a simpler memory test plugin to speed up the tests) detects it.

In 94.7% of the 262,016 experiments, RAMpage succeeded in claiming the faulty page frame, testing the memory and detecting the fault. The remaining memory areas (see fig. 4) primarily host the kernel image or its data structures, most prominently the MMU page tables, or x86 legacy such as the "640 kiB hole" (also cf. subsection 5.2). In 0.2% of the experiments, RAMpage's diagnostic output did

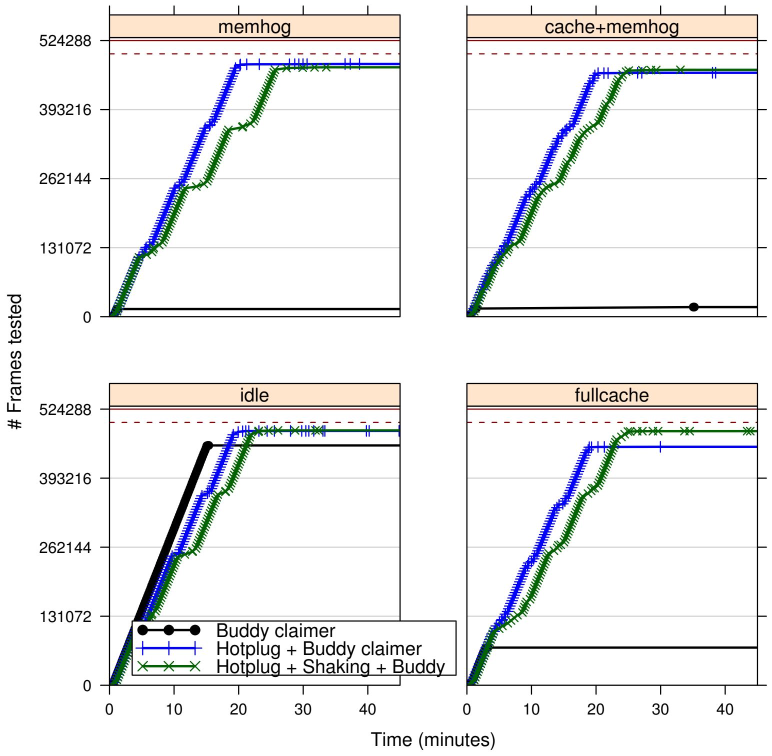

Figure 5 Page-frame claim rates for the different claiming methods in four load scenarios. The dashed red line indicates the maximum amount of testable pages in the 2 GiB address range (499,082 of 524,288 pages).

not show any progress after injecting the fault: We conjecture this to be a symptom of the faulty memory address being vital for the correct operation of Linux or RAMpage itself.

For proving the usability of our testing framework with *real* broken memory hardware, we conducted experiments on a two-way AMD Opteron 250 server with eight 1 GiB DDR1-400 modules (Apacer ECC Registered DDR-400, with ECC disabled in the BIOS), one of which was known defective. Memtest86+ runs confirmed a vast amount of single- and multi-bit errors throughout the module's complete address range; after further manual analysis we hypothesize that either the buffer chip or one of the 16 memory chips on that module is causing the faults.

RAMpage successfully detected all defective page frames from the faulty DIMM and removed them from further use. After offlining the problematic address range, the machine continued running smoothly with the remaining 7 GiB of memory. Placing the defective DIMM in the "wrong" memory slot on the mainboard led to Linux failing to boot – an effect expected when essential kernel data structures or machine instructions are placed in the affected address range.

5.2 Efficiency: Throughput and Memory Coverage

To assess RAMpage's page-frame claiming efficiency (i.e., memory coverage) and testing throughput, we logged the progress of the three page-frame claiming methods over time in four different scenarios (fig. 5):

- A freshly booted, idle machine with most frames unused (*idle* in fig. 5),
- a full buffer cache by reading a large amount of files beforehand (*fullcache*),
- a single process hogging all memory and initially writing data to it to circumvent memory overcommitment (*memhog*),
- and a combination of the last two scenarios (*cache+memhog*).

About 4.8% of all frames (∼98.5 MiB) are considered "untestable" right from the start: These page frames, marked with the *reserved* flag, are not even touched by other kernel subsystems such as, e.g., memory hotplugging, and kernel source comments advise not to tamper with these. A closer look revealed that the majority of these frames hold the kernel itself and its data structures (cf. fig. 4).

The experiments indicate that in most scenarios, a large percentage of the remaining 499,082 page frames can be claimed and tested. The buddy claimer fails to claim frames that are in use by the buffer cache or user-space applications. The hotplug and shaking methods prove significantly more effective for these frame types, at different levels of success. Since our earlier experiments with Linux 2.6.35 (Schirmeier et al. (2011)), the kernel's memory management subsystem seems to have improved substantially: In the old version, the *combination* scenario showed a suboptimal memory coverage; with Linux 3.5 it is on par with the other scenarios.

The test throughput (using the unthrottled "full-speed" test scheduler) is sufficient to test the evaluation PC's 2 GiB of RAM in well under 30 minutes, which emphasizes RAMpage's practicality. As outlined in subsection 3.3.2, one pass can be stretched to a longer period of time (e.g., 24 hours) with the "slow" scheduler, still re-testing the memory in a reasonable time window.

5.3 Practicality: Impact on Performance, Latency and Energy Consumption

RAMpage's impact on the target machine's power consumption, or the performance or latency of concurrently running software (system services, user applications, whatever the machine's primary purpose should be) is of high practical relevance. If the test is qualitatively sound but disturbs normal operations severely, the intended purpose to replace offline testing is put into question. Also, an unreasonably high increase in energy consumption may render the approach too costly to apply on a regular basis.

We chose a set of common benchmark scenarios:

- Linux 2.6.35 build ("allmodconfig"), highly parallelized ("make -j8"), known for high CPU and I/O loads, and memory consumption.
- POV-Ray (v3.6.1, run in benchmark mode), a single-threaded CPU benchmark with moderate memory requirements and close to zero I/O.

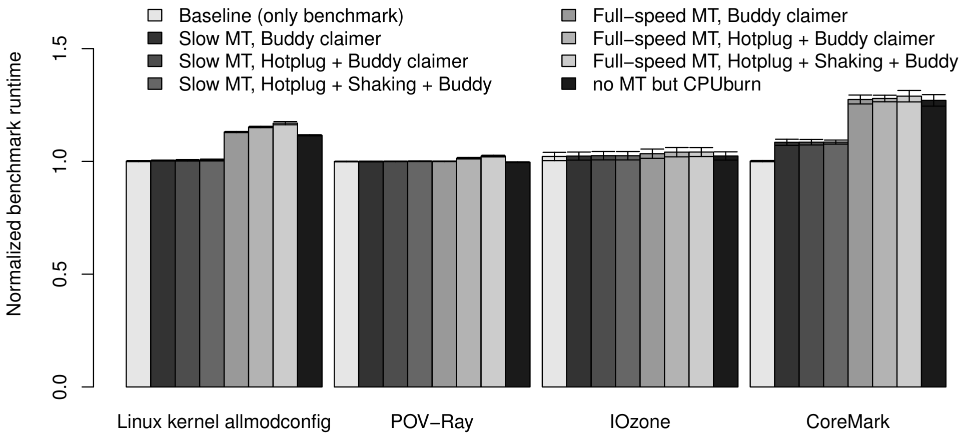

Figure 6 Effects of various online memory testing schedulers and frame claimers on *Linux 2.6.35 building* (allmodconfig), *POV-Ray*, *IOzone* and *CoreMark* benchmarks (normalized averages, with error bars indicating SD).

- IOzone (r3.308, run with write/rewrite and read/reread tests, a file size of 1,500 MiB, and block sizes of all powers of two between 4 kiB and 512 kiB), an I/O-bound filesystem benchmark expected to be sensitive to buffer-cache contents being dropped. Note that the chosen file size is – contrary to IOzone's recommendations for real hard-disk benchmarking – *smaller* than the available RAM, as we want to exhibit buffer-cache effects.
- CoreMark (v1.0, with default settings, running in parallel with four threads)[1], a CPU-bound multi-core benchmark.
- tbench (v4.00, run with a single client on the same machine), a file-server benchmark stripped of the actual server-side operations, yielding a pure TCP socket benchmark for latency measurements.

Note that only Linux build and CoreMark utilize all CPUs in the test machine. We presume that load scenarios *not* occupying all available resources are relatively common.

Fig. 6 shows normalized benchmark runtime averages with a baseline measurement, runtimes with the memory tester in action – comparing the three page frame claimers and the "slow" with the "full-speed" test scheduler – and measurements without the memory tester but a single-threaded CPU-consuming process ("CPUburn"). Notable are several observations:

- The "slow" test scheduler, aiming at reasonable memory test cycles of 24 hours, has no significant impact on any of the benchmarks. A system's normal operation is not disturbed, allowing to employ RAMpage permanently.
- Especially the kernel build and CoreMark are hit by the full-speed memory tests, in particular when the hotplug and shaking methods for frame claiming are in use. The slowdown is partially due to the CPU cycles the Memtest86+ tests consume (compare to the CPUburn measurements which resemble this

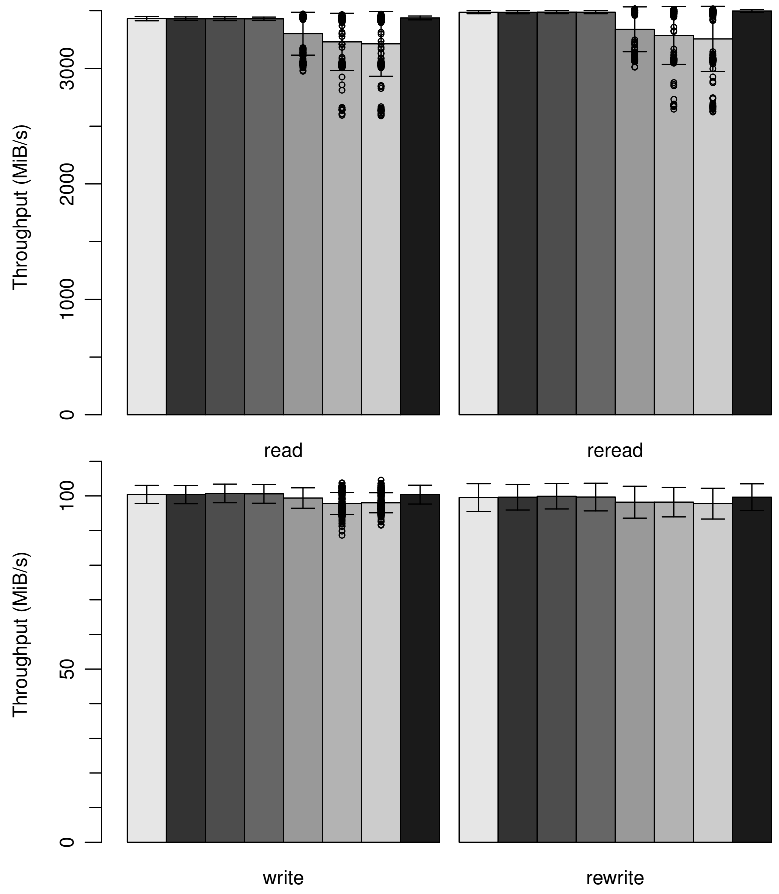

Figure 7 Memory test effects on *IOzone* “read”, “reread”, “write” and “rewrite” benchmark throughput averages (with error bars indicating SD), with individual measurements for the full-speed scheduler with different claimers.

effect). Additional penalties supposedly come from a high memory bus load (all test schedulers) and – especially for the kernel build – the buffer-cache pages that need to be reloaded from hard disk after they have been dropped for testing (hotplug and hotplug+shaking claimers, the latter being even more aggressive towards buffer-cache frames – cf. subsection 5.2).

- The single-threaded, CPU-bound POV-Ray benchmark is not affected at all.

- IOzone’s runtime seems not to be affected either, but a closer look at its *writing* and most notably its *reading* benchmarks (fig. 7) reveals that – as expected from the claiming methods’ behaviour already described – the hotplug and shaking methods *in some test runs* affect the reading throughput. The scattering of results can be explained by some runs with exactly the wrong (still to be read by the benchmark) or the right (already read and not needed anymore) buffer-cache frames being dropped.

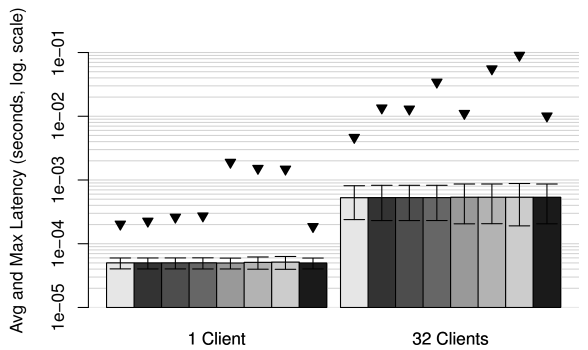

Figure 8 Memory test effects on average (with error bars indicating SD) and maximum latency for the *ReadX* operation in the *tbench* benchmark with 1 and 32 clients connected to a server running on the same host (Y axis in logarithmic scale; colour coding legend in fig. 6).

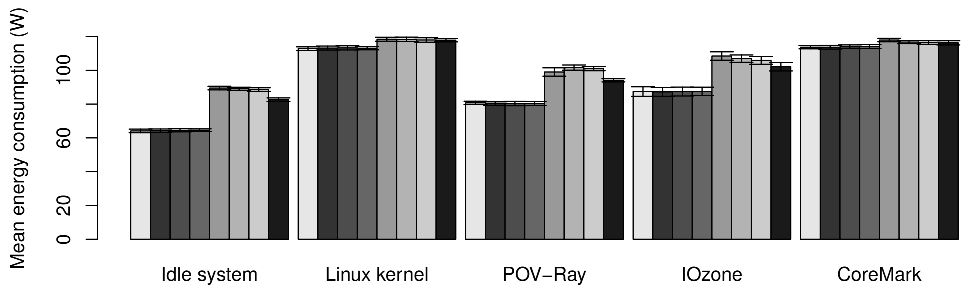

Figure 9 RAMpage's impact on the test machine's energy consumption: At "slow" testing speed, the increase is negligible.

Fig. 8 shows average and maximum latencies for the *ReadX* operation (which we picked to get comparable numbers as latencies differ by a few percent among the different operations) of the *tbench* benchmark talking via TCP to a server running on the same host. Although the average latencies do not vary significantly among the different claiming and test scheduling strategies, the *maximum* latencies go up by a factor of 9 (32 clients: factor of 19) from the baseline measurement to the most intrusive claimer variants. As memory bus and CPU load stays the same for all measurements with memory testing, only the claiming method itself can be blamed for these rare spikes. As the memory hotplug subsystem was never intended for continuous use but occasional system maintenance events, it was designed for safety rather than for high performance, which we conjecture to be causing the latency extrema.

To assess the additional energy consumption of our test machine, we hooked up a high-quality consumer standby-energy measurement device (NZR SEM 16+ USB) to its power supply. For each single measurement, we ran one benchmark and claimer combination for one hour and recorded the absolute energy consumption. Each combination was measured at least ten times.

With RAMpage running the "slow" memory test, the additional energy consumption is negligible (fig. 9) – a good indication that it could run on large server farms without having to fear an unpleasant impact on operational costs. At full speed, the average power takeup of our test machine ramps up from 60 W to about 90 W on an otherwise idle system; with approximately 10% more consumption than the CPU-centric CPUburn run, the additional stress on the memory subsystem becomes evident. The more energy-hungry the benchmarks get, the less repercussions a concurrent full-speed memory test seems to have: But, of course, this needs to be taken with more than one grain of salt, as the total benchmark runtime increases (cf. fig. 6), and fig. 9 only shows the *mean* energy consumption.

6 Related Work

We consider related work from three distinct areas, software-based online and offline memory tests, hardware-based methods for memory error detection and correction, and operating-system integrated EDAC frameworks.

6.1 Memory Tests

Memtest86+ (Demeulemeester (2011)) is a bare-metal offline memory tester for x86-based systems using continuous tests with different patterns and algorithms. Memory used by the tester is tested in advance, enabling a 100% coverage. In addition to tests under normal operating conditions, stress testing memory outside its specifications is possible by adjusting refresh rates and disabling ECC.

Singh et al. (2005) describe an online memory test for Solaris, which is kernel-based and sequentially tests memory ranges by applying the fault model of Nair et al. (1978). A simple frame scheduler calculates memory ranges to be tested based on configured allocation size and test iteration number. Defective frames are excluded from further use, and processes that accessed these frames are killed. Results obtained by installing known faulty memory modules and running two different workloads showed that detected faults differed between experiments. Only about one third of the detected errors overlapped. (This correlates with Schroeder et al. (2009), which state that system utilization has a strong influence on the number of memory errors detected by ECC hardware.) This system superficially resembles RAMpage. However, the pure kernel-mode implementation significantly reduces flexibility. In addition, running on SPARC eliminates many of the problems we had to face with hardware variability in x86 systems. Singh et al. did not publish performance values, which we consider essential.

Effo GPL is a Linux user-space memory tester that implements several of Memtest86+'s fault models. It acquires frames by allocating all obtainable memory, using a kernel module to translate virtual to physical addresses. Unfortunately, the publication that described the program and test algorithms has been removed from the program's website. Similar implementations of purely user-space memory tests exist, such as Memtester by Cazabon (2009).

Similar to a bare-metal tester, Effo monopolizes system memory. Since it cannot directly acquire specific page frames from user space, it has to allocate many frames and hope that the desired frame is included. This is a serious shortcoming for a memory test that should not severely degrade system performance. Additionally, this approach cannot predict which frames are to be allocated.

Rahman et al. (2011) describe COMeT, another online memory tester for Linux that was developed in parallel and independently of RAMpage. Their approach is similar in idea to ours, though the efficiency of their implementation lags behind, causing a performance degradation of up to a factor of 4.41, and much lower page-frame claim rates. However, the authors designed a sophisticated scheduling approach: It guarantees that any page frame given out to a memory-demanding application was tested *before*, and is re-tested in specific time intervals. A pool of freshly tested page frames keeps allocation latencies low and is replenished at rates driven by the current system load and memory allocation demands. Combining RAMpage's efficient page-frame claiming approaches with COMeT's scheduling methods could be an interesting prospect.

Amvrosiadis et al. (2012) describe a related analysis of a novel load-aware approach for hard-disk scrubbing, which considers timing and size of scrubbing related to other system activity. Compared to RAM testing, this approach has to consider different parameters, like rotational delay of hard disks and seek times; the basic idea seems similar. An integration of RAM and hard-disk background testing in a common future EDAC framework seems desirable.

6.2 Fault Management

Solaris 10 What's New: Predictive Self-Healing (n.d.) describes a framework to improve system reliability using a *fault manager* that interprets hardware errors. If a faulty component is detected, it tries to offline that component. The system, described by Romack (2006), consists of components to manage services, to enrich fault logs, and to provide fault prediction in cases where dependencies are well-known. This implementation is one of the most advanced fault management systems in commodity operating systems. However, it requires a large amount of hardware add-ons and generally operates on a more coarse-grained basis compared to our memory-testing approach.

The Linux EDAC framework (Thompson (2010)) is used to handle (usually ECC memory) hardware errors. It provides an abstraction layer representing the physical layout of memory in modules, down to a module's chip-select rows. This requires dedicated drivers for specific chipsets. The framework can also detect ECC errors in non-RAM components, such as buses, DMA engines, caches, etc. EDAC is not CPU-architecture specific; currently, drivers for x86 and PPC architectures exist. While EDAC is useful for ECC-equipped systems, its functionality relies heavily on hardware support for error detection, making it unsuitable for commodity systems.

Linux can log CPU-specific machine check exceptions, such as correctable and uncorrectable errors, using the mcelog user-space tool (Kleen (2010)). The kernel itself performs immediate actions, like killing processes. While mcelog proves to be a useful tool for system administrators, again, it requires the existence of specific hardware, i.e., MCE-enabled x86 CPUs. In addition, it does not provide methods to handle errors.

6.3 Hardware-based approaches

In an IBM whitepaper, Dell (1997) discusses shortcomings of SECDED (Single Error Correction, Double Error Detection). ECC multi-bit errors, caused by a single memory chip failing completely, are very likely. However, SECDED cannot handle these.

Chipkill is a mechanism designed to survive such failures (Chen (2001), Olarig (2003), Yoon & Erez (2010) and Dell (1997)). It expands the granularity of memory accesses, protecting against the failure of a complete memory chip. Sun UltraSPARC-T1/T2 (Sun Microsystems (2008)) and AMD Opteron (AMD, Inc. (2007)) systems implement this strategy by accessing two memory modules simultaneously. While Chipkill is a useful extension to general ECC-based protection mechanisms, it is only employed in expensive high-end server systems.

The AMD K8 architecture (AMD, Inc. (2008)) introduced memory scrubbing. The CPU can continuously read memory in order to provoke ECC errors early, i.e., as long as only correctable single-bit errors show up. However, this technique is only useful for ECC-equipped systems, since the scrubbing is performed read-only. According to AMD's documentation, currently DRAM scrubbing is unsupported, which restricts hardware-supported scrubbing to cache memory.

Solaris x86 supports memory scrubbing. Like AMD's hardware scrubber, this feature is only useful on ECC-equipped systems. Of interest is a section in the scrubbing code (Sun Microsystems (n.d.)) defining a 12-hour testing interval for the complete physical memory of a system, commenting "twice the frequency the hardware folk estimated would be necessary".

7 Discussion and Future Work

Although the overall design of RAMpage follows a clean and well-structured approach, attaining an implementation that fulfilled our functionality and efficiency goals was hampered by numerous small problems. Most of these problems were rooted in assumptions the Linux kernel makes about the underlying memory system.

Understanding, partially reverse-engineering and extending the complex and convoluted Linux memory-management system turned out to be one of the less trivial undertakings – not least due to the lack of up-to-date documentation for the related subsystems. We required several experimentation iterations until we procured a reliable method of claiming page frames (especially the ones currently in use). Respecting Linux' own limitations regarding certain reserved memory areas – e.g., its inability to migrate the kernel image or virtual-memory data structures to other areas in physical memory – we arrived at a reasonably stable, best-effort solution that nevertheless is able to improve the reliability of a commodity Linux-based x86-64 system significantly. Especially porting RAMpage to Linux 3.5 removed some of the oddities we observed earlier, resulting in substantially better page-frame claim rates in all benchmark scenarios, compared to our previous publication.

We presume that a tighter integration with Linux' memory management would allow to test an even higher percentage of the overall physical memory. The complexity of Linux memory management, as, e.g., described by Gorman (2004), and the high rate of changes, however, make this seem a daunting task. Also an integration with a Linux-based fault management framework (such as EDAC) might be worthwile.

Despite all the positive properties of our graceful degradation approach, some general drawbacks of an online memory test shall not remain unmentioned. Overcoming these limitations is an interesting topic for future research.

An inherent limitation of any online memory-testing approach is that all program execution *in-between* a successful check of a page frame and a subsequently found error therein may be affected by that error. We doubt this can be remedied by any reasonably efficient software solution. Thus, the current solution only reduces the probability of an application using a defective memory cell, but cannot avoid the situation completely.

One possible approximation to further minimise this probability is to test page frames *before* giving them out to any process or kernel driver *for the first time*. Rahman et al. (2011) developed an elaborate pre-allocation test scheduler for their online memory tester, COMeT (cf. subsection 6.1): A pool with freshly-tested page frames keeps allocation latencies low and is replenished driven by the application's current allocation demands. These page frames can also be used as the migration target for the content of frames about to be tested. RAMpage currently delegates migration to Linux hotplugging (cf. subsection 4.2), which picks an arbitrary free page frame for migration instead of taking test "aging" into account. Combining RAMpage's superiour page-frame claiming methods and low impact on normal operations, and COMeT's demand-driven pre-allocation approach, should result in an even better tool for everyday use.

An idea complementary to the kernel-based RAMpage approach outlined in this paper is to position the memory testing infrastructure *below* the running OS kernel. This could be achieved by employing a hypervisor like Xen, which provides kernels running on top with the illusion of running on a physical machine. Advantages of this approach would be the possibility to test all of a (virtual) machine's memory and to gain a certain level of operating system independence. A possible drawback would be the more complex installation and configuration of such a system. Since ever more server systems are being based on virtualization, however, this might not cause significant overhead in these systems.

Finally, it would be interesting to see how well our online memory tester is portable to different architectures (especially embedded systems, considering the inherent limitations of these systems), or even to other Unix-like systems, such as BSD, Solaris, or MacOS X. The primary prerequisites for a port are MMU-based OS services for page-frame claiming and migration (such as those identified in Linux, cf. subsection 6.1), and a mechanism to exclude frames from further use (such as Linux *hwpoison*).

References

AMD, Inc. (2007), 'BIOS and kernel developer's guide for AMD NPT family 0fh processors', `http://support.amd.com/us/Processor_TechDocs/32559.pdf`.

AMD, Inc. (2008), 'BIOS and Kernel Developer's Guide (BKDG) for AMD Family 11h Processors, rev 3.00'.

Amvrosiadis, G., Oprea, A. & Schroeder, B. (2012), Practical scrubbing: Getting to the bad sector at the right time, *in* 'Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)', IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 1–12.

Bonwick, J. (1994), The slab allocator: An object-caching kernel memory allocator, *in* 'Proceedings of the 1994 USENIX Annual Technical Conference', USENIX Association, Berkeley, CA, USA, pp. 6–17.

Cazabon, C. (2009), 'memtester website'.
URL: *http://pyropus.ca/software/memtester/*

Chang, M.-F., Fuchs, W. K. & Patel, J. H. (1989), 'Diagnosis and repair of memory with coupling faults', *IEEE Transactions on Computers* **38**, 493–500.

Chen, C.-L. (2001), 'Symbol level error correction codes which protect against memory chip and bus line failures', US Patent 7093183.
URL: *http://www.freepatentsonline.com/7093183.html*

Dell, T. J. (1997), A white paper on the benefits of chipkill-correct ECC for PC server main memory, *in* 'IBM Whitepaper'.

Demeulemeester, S. (2011), 'Memtest86+'.
URL: *http://www.memtest.org/*

Gorman, M. (2004), *Understanding the Linux Virtual Memory Manager*, Prentice Hall.

Hayes, J. P. (1975), 'Detection of pattern-sensitive faults in random-access memories', *IEEE Transactions on Computers* **24**(2), 150–157.

Kleen, A. (2010), mcelog: memory error handling in user space, *in* 'Proceedings of the 2010 Linux Kongress'.

Nair, R., Thatte, S. & Abraham, J. (1978), 'Efficient algorithms for testing semiconductor random-access memories', *IEEE Trans. on Computers* **C-27**(6), 572 –576.

Nightingale, E. B., Douceur, J. R. & Orgovan, V. (2011), Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs, *in* 'Proceedings of the ACM SIGOPS/EuroSys International Conference on Computer Systems 2011 (EuroSys '11)', ACM Press, New York, NY, USA, pp. 343–356.

Olarig, S. P. (2003), 'Technique for implementing chipkill in a memory system', United States Patent 7096407.
URL: *http://www.freepatentsonline.com/7096407.html*

Panzer-Steindel, B. (2007), Data integrity, Technical report, CERN, Geneve, Switzerland.

Rahman, M., Childers, B. R. & Cho, S. (2011), COMeT: Continuous online memory test, *in* 'Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '11)', IEEE Computer Society Press, pp. 109–118.

Romack, R. (2006), *Service Management Facility (SMF) in the Solaris 10 Operating System*, Sun, Part No 819-5150-10.

Schirmeier, H., Hoffmann, M., Kapitza, R., Lohmann, D. & Spinczyk, O. (2012), FAIL*: Towards a versatile fault-injection experiment framework, *in* G. Mühl, J. Richling & A. Herkersdorf, eds, '25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings', Vol. 200 of *Lecture Notes in Informatics*, German Society of Informatics, pp. 201–210.

Schirmeier, H., Neuhalfen, J., Korb, I., Spinczyk, O. & Engel, M. (2011), RAMpage: Graceful degradation management for memory errors in commodity linux servers, *in* 'Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '11)', IEEE Computer Society Press, Pasadena, CA, USA, pp. 89–98.

Schroeder, B., Pinheiro, E. & Weber, W.-D. (2009), DRAM errors in the wild: A large-scale field study, *in* 'Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems', SIGMETRICS '09, ACM, New York, NY, USA, pp. 193–204.

Singh, A., Bose, D. & Darisala, S. (2005), Software based in-system memory test for highly available systems, *in* 'Proceedings of the 2005 International Workshop on Memory Technology, Design, and Testing (MTDT '05)', IEEE Computer Society, Washington, DC, USA, pp. 89–94.

Solaris 10 What's New: Predictive Self-Healing (n.d.).
URL: *http://docs.oracle.com/cd/E19253-01/817-0547/esqej/index.html*

Srini, V. P. (1978), 'Fault location in a semiconductor random-access memory unit', *IEEE Transactions on Computers* **27**, 349–358.

Sun Microsystems (2008), 'OpenSPARC T2 system-on-chip (SOC) microarchitecture specification'.

Sun Microsystems (n.d.), 'memscrub.c source code, `http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/i86pc/os/memscrub.c`'.

Thompson, D. (2010), 'Linux kernel documentation: EDAC – error detection and correction', [Linux 2.6.35]/Documentation/edac.txt.

Yoon, D. H. & Erez, M. (2010), 'Virtualized and flexible ECC for main memory', *ACM SIGARCH Comput. Archit. News* **38**, 397–408.

Notes

[1]For a meaningful comparison with the other benchmarks, the plot shows the *normalized reciprocal* of the CoreMark score. The complete report with parameters and average score for baseline, according to CoreMark reporting rules: `CoreMark 1.0 : 32943.657830 / GCC4.4.5 -O2 -DMULTITHREAD=4 -DUSE_PTHREAD -DPERFORMANCE_RUN=1 -lrt / Heap / 4:PThreads`