RAMpage: Graceful Degradation Management for Memory Errors in Commodity Linux Servers

Horst Schirmeier, Jens Neuhalfen, Ingo Korb, Olaf Spinczyk, and Michael Engel
Department of Computer Science 12, Technische Universität Dortmund
Otto-Hahn-Str. 16, 44221 Dortmund, Germany
Phone: +49-231-755-6142, Fax: +49-231-755-6116
e-mail: {horst.schirmeier, jens.neuhalfen, ingo.korb, olaf.spinczyk, michael.engel}@tu-dortmund.de

***Abstract*—Memory errors are a major source of reliability problems in current computers. 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.**

Often, neither additional circuitry to support hardware-based error detection nor downtime for performing hardware tests can be afforded. In the case of permanent memory errors, a system faces two challenges: detecting errors as early as possible and handling them while avoiding system downtime.

To increase system reliability, we have developed RAMpage, an online memory testing infrastructure for commodity x86-64-based Linux servers, which is capable of efficiently detecting memory errors and which provides graceful degradation by withdrawing affected memory pages from further use.

We describe the design and implementation of RAMpage and present results of an extensive qualitative as well as quantitative evaluation.

***Keywords*-Fault tolerance, DRAM chips, Operating systems**

I. Introduction

Operating an internet server for business-oriented purposes requires high availability, often combined with low-cost demands. The operator of such a server faces a tough decision – is it more harmful to a business to operate a server without downtime as long as possible while ignoring the risk of possible memory errors, or to shut down the server in regular intervals to perform lengthy memory tests?

A recent study performed by Bianca Schroeder et al. at Google [1] analyzed the reliability of off-the-shelf memory DIMMs (Dual In-Line Memory Modules) in a large percentage of Google's server farm. The result of this analysis is eye-opening – it turns out that permanent memory errors are an order of magnitude more common than previously assumed. On average, a permanent error is expected to show up once a year in a standard 1 GiB DDR memory module. Thus, when operating a small server with several modules, a single-bit error may show up once every few months.

In the best possible case, this error causes a program or operating system crash or malfunction and is comparatively easy to detect. However, an analysis of errors in CERN's systems [2] showed that the probability of *silent data corruption* errors – i.e., errors that stay undetected and result in unexpected modification of some value in a running program with possibly severe consequences for the application and its users[1] – is several orders of magnitude higher than expected by the failure statistics for the affected components.

This observation is also supported by a recent study by Nightingale et al. [3], 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.[2]

For cost-sensitive systems, the probability of an undetected permanent memory error has to be reduced. Memory tests can help to detect errors; if memory 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+ [4], is in most cases unacceptable. Thus, in order to avoid system downtime, we developed an approach to run a memory tester as a system service during normal operation.

In this paper, we describe RAMpage, our system to perform proactive online memory tests for commodity x86-64 Linux systems that enables the operating system to perform graceful degradation in case a memory error is detected. Compared to existing memory test mechanisms, our approach requires no downtime and allows for flexible selection of memory test frequency and test methods by the system operator. Memory tests run in user space, only physical memory allocation and – in case of a detected error – degradation handling is performed by a kernel module.

The paper is organized as follows. Section II discusses graceful degradation and its realization in Linux, followed by an overview of testing methods in section III. The structure of RAMpage and details of its implementation are described in sections IV and V. The effectiveness and efficiency of in-system memory tests are evaluated in section VI. Section VII discusses related work; finally, section VIII concludes the paper and gives an outlook to ideas for future research.

[1] E.g., consider a flipped sign bit in a money transfer ...

[2] This is actually in contrast to the findings by Schroeder et al. [1]. We expect further investigations to clear up this confusion.

II. Graceful Degradation

A system which cannot be shut down for maintenance due to availability requirements or other circumstances, like operating a system at a remote site, must provide some way to handle components that fail during runtime. This so-called *graceful degradation management* provides a best-effort approach to operate a system as long as possible, albeit potentially with reduced functionality or performance.

Approaches that implement graceful degradation can be realized on various levels of the system architecture – in a multi-processor system, a defective CPU could be turned off and processes migrated away from that processor, in a system using multiple memory banks, one of the memory banks or chips could be disabled [5], and a system with multiple network connections may be able to change routing tables in case one of the interfaces fails.

In the case of memory faults, however, disabling a complete memory DIMM implements a rather coarse-grained level of degradation. If more than one DIMM is available after all, still a significant portion of the overall memory would be disabled. Virtual memory allows to control physical memory use on a finer granularity – the size of a page frame, typically a few kilobytes in size, depending on the architecture. However, operating system support for removing pages from a system's address space is required to achieve this fine granularity.

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. However, the framework currently 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.

```
             total       used       free  shared  buffers
Mem:      33017208   31062408    1954800       0  3774740
-/+ buffers/cache:  8007848  25009360
Swap:      3863592      75384    3788208
```

Figure 1. Memory use in a typical Linux server system, provided by the *free* command-line utility.

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 V.

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. Additionally, *hwpoison* provides another method for claiming allocated page frames, which makes it even more suitable for our memory tester.

III. Memory Testing

Memory testing is a rather well-explored area of research. Examples of test algorithms can be found in early papers by Nair [6] and Hayes [7] as well as in more recent publications [8], [9], [10]. 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 VII-A) 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. The RAMpage testing framework thus has to ensure this for the memory area requested by the testing scheduler before testing can commence.

In early Unix systems, the amount of memory used by the kernel was rather fixed, e.g., the size of the buffer cache for disk accesses was configured as a compile- or link-time kernel

parameter. Thus, if few applications were running on a system, a considerable amount of memory would lie unused. Modern Unix-like systems, however, tend to use up almost all physical memory for the (unified) buffer caches and VM cache. Fig. 1 shows a snapshot of a Linux system equipped with 32 GiB of RAM. Of the total amount of memory available, about 25 GiB are actually used for caching.

A memory tester for a current system 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 may be assigned to specific physical addresses in the peripheral's configuration by a device driver; MMU page tables in main memory are usually not relocatable. In future systems, IOMMUs [11] might solve the DMA problem. However, a certain amount of memory will still be untestable since it is reserved by the kernel. Employing virtualization approaches like Xen or KVM seems to be a good approach to test all the memory used by a *virtual* machine. However, this still does not provide access to all of the *physical* memory of a system and, thus, will not necessarily increase reliability – there is no simple way to test memory the virtual machine monitor itself has allocated.

Another question influencing the design of RAMpage is how many pages need to be tested concurrently. All pages that are to be tested at the same time, e.g., in order to find correlation errors, are not available to the rest of the system during the test. In general, a tester should only allocate as many pages as required and release pages as soon as the test is finished in order to minimize the testing overhead.

However, the specific structure of Linux memory management and other, non-functional requirements to memory testing pose additional restrictions, which are discussed in detail in the following section.

IV. 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 VII for a detailed discussion). In the following paragraphs, 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.

A. Design Considerations

The major intention underlying the design of RAMpage was to provide a solution to perform online memory tests in 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.

A useful memory tester has to be able to cover as much physical memory as possible. Special care has to be taken to ensure that the real, physical RAM content is actually tested and that the tests operate on physical addresses, so a faulty component can be identified easily.

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 V) 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, are handled in a user-mode component.

Since our aim was not to reinvent memory testing, the RAMpage user-space component does not perform memory tests itself, but delegates them to a flexibly exchangeable memory test plugin. The 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. In the current version described in this paper, RAMpage provides ports of all tests that are available in the Memtest86+ [4] code base as memory test plugins (cf. fig. 4).

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 paragraph.

B. 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.[3] While a pure user-space–based testing approach would be preferable – no kernel modifications are required, reducing critical in-kernel code to zero – the preconditions for *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

[3]Using root privileges, general access to physical memory is possible in Linux using the /dev/mem special device. However, outside of the kernel no information is available as to the current use of a specific page frame.

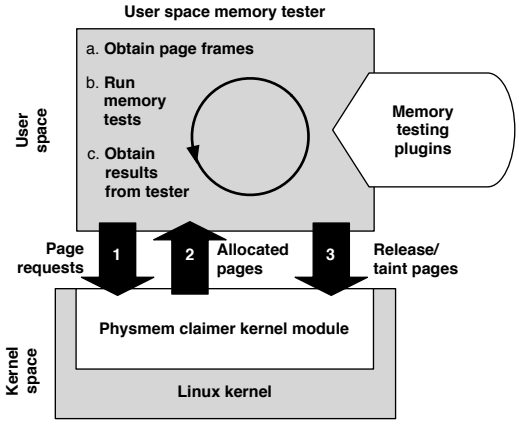

Figure 2. RAMpage's overall structure: The user-space memory tester with exchangable testing plugins consults the *physmem* kernel module to obtain physical page frames, and to taint frames positively tested for defects.

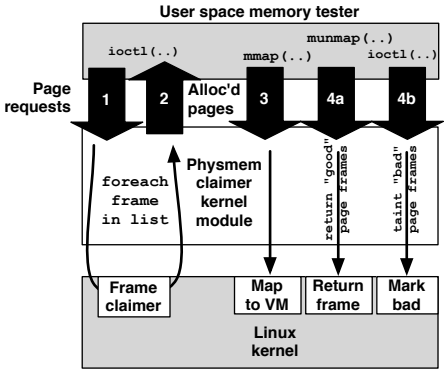

Figure 3. RAMpage's *physmem* page frame claimer kernel module and its system call interface.

Linux VM system does not even guarantee that the logical address space is backed by physical memory at every point in time.[4] 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; one example for such a tester is Effo, described in detail in section VII.

In consequence, a memory tester has to be able to request *physical* memory page frames in order not to repeatedly re-test a small memory area. Knowledge of physical addresses is also needed 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.

C. High-Level Structure

RAMpage's overall structure is shown in fig. 2. 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.

D. System Components

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 V 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 squealed 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. 3. 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).

E. Test Scheduling

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, 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

[4]Still worse, Linux supports *memory overcommitment*, allowing processes to allocate more memory than actually physically available in the system, including swap space. A memory tester that allocates huge amounts of virtual address space may thus fall victim to Linux's out-of-memory killer.

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
6	Moving inversions, 32-bit pattern
7	Random number sequence
8	Modulo 20, random pattern

Figure 4. Memtest86+ [4] tests ported to RAMpage. Test #0 cannot be run effectively as it needs access to a large fraction of physical memory at the same time.

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).

V. 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 2.114 lines of C code, whereas the user-space is implemented in 1.447 lines of Python code. The memory tester plugins, a port of the well-established Memtest86+ [4] tests (fig. 4), add up to 867 lines of C code. One limitation is that the address line test #0 cannot be run effectively in our infrastructure, as it needs access to a large fraction of physical memory at the same time. This would defeat our design goal of minimal-impact online testing.

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

A. 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 actually 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 [12]) 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.

B. Claiming Page Frames

RAMpage's userland 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 userland 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 varies 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 VI). 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.[5] 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.

C. 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 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

[5]Note that this excludes the destructive part of *hwpoison* that resorts to killing processes when claiming fails (cf. section II).

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.

D. Portability

In principle, our implementation can easily be ported to other hardware architectures, such as i386 or PPC. Practical limitations arise from the fact that Linux currently does not implement memory hotplugging and/or *hwpoison* for all possible architectures. In those cases, therefore, either these subsystems would need to be ported, too, or other means to claim used page frames would have to be discovered.

Porting RAMpage to another operating system would require rewriting the kernel module specifically for the target-system's memory-management internals. The user-space component including existing test plugins would remain unchanged, aside adaptations to another OS kernel interface.

VI. 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 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 2.6.35 kernel[6]. The following paragraphs describe the experiments we conducted and the conclusions we draw.

A. Effectiveness: Virtually and Really Broken Hardware

In order to assess RAMpage's effectiveness, we conducted a series of automated tests with FAUmachine [13], a virtual machine environment specifically designed for hardware-error injection. FAUmachine can be configured to inject single-bit errors at arbitrary memory locations.

Well-reproducible results showed that randomly chosen single-bit stuck-at faults are detected and reported reliably (unless the guest OS fails to boot and/or start the test at all); subsequently the testing framework successfully removed the frame from the memory subsystem. A small amount of test runs did not locate the error, most probably due to the fault being located in a frame our kernel module is unable to claim (cf. subsection VI-B).

[6]Linux was slightly modified in order to export some internal symbols needed by our module, and to timeout faster when trying to hotplug memory.

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.

B. Efficiency: Throughput and Memory Coverage

In order 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,
- a *full buffer cache* (by reading a large amount of files beforehand),
- a single process *hogging all memory* and initially writing data to it to circumvent memory overcommitment,
- and a *combination* of the last two scenarios.

About 4.5% of all frames (∼92 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.

The experiments indicate that in most scenarios, a large percentage of the remaining 500,807 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. We cannot give a good explanation for the suboptimal memory coverage in the last scenario – a full buffer cache, evicted by a user-space memory-hogging application – and can only conjecture a quirk in the kernel subsystems we utilize for claiming.

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 IV-E, one pass can be stretched to a longer period of time (e.g.,

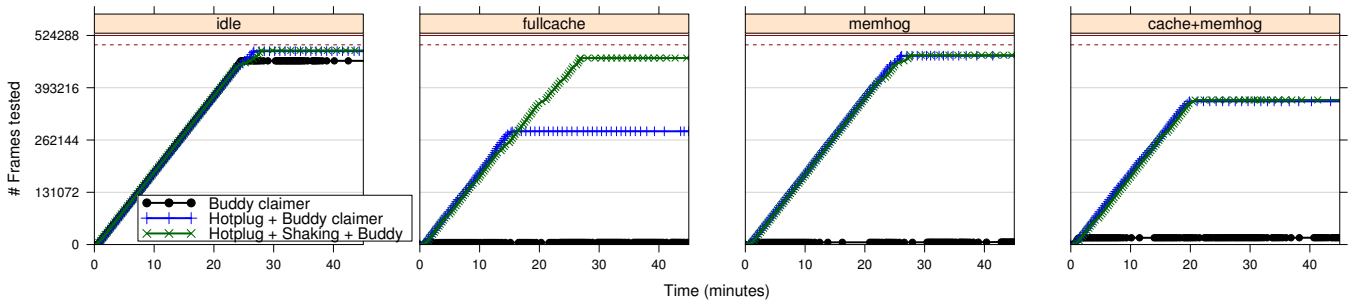

Figure 5. Page-frame claim rates for the different claiming methods in four load scenarios. The dashed red line indicates the maximum amount of testable pages in the 2 GiB address range (500,807 of 524,288 pages).

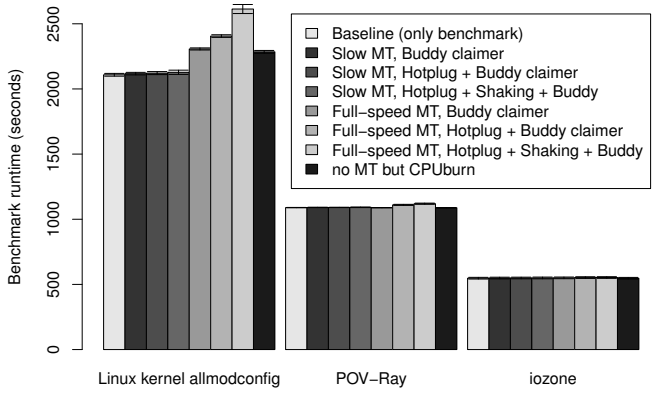

Figure 6. Effects of various online memory testing schedulers and frame claimers on *Linux 2.6.35 building* (allmodconfig), *POV-Ray*, and *IOzone* benchmarks (averages, with error bars indicating SD).

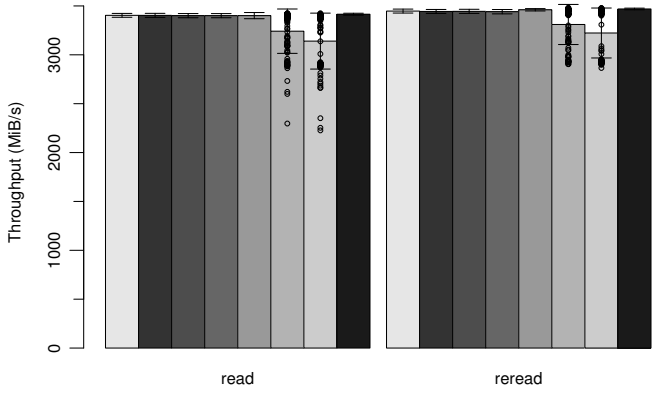

Figure 7. Memory test effects on *IOzone* "read" and "reread" benchmark throughput averages (with error bars indicating SD), with individual measurements for the full-speed scheduler with hotplug and shaking claimers.

24 hours) with the "slow" scheduler, still re-testing the memory in a reasonable time window.

C. Practicality: Impact on Performance and Latency

RAMpage's impact on 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.

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.
- 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.
- 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 one benchmark (Linux build) utilizes all CPUs in the test machine. We presume that load scenarios *not* occupying all available resources are relatively common.

Fig. 6 shows 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 regularly.
- Especially the kernel build is 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 effect). Additional penalties supposedly come from a high memory bus load (all test schedulers) and the buffer cache pages that need to be reloaded from hard disk after they have

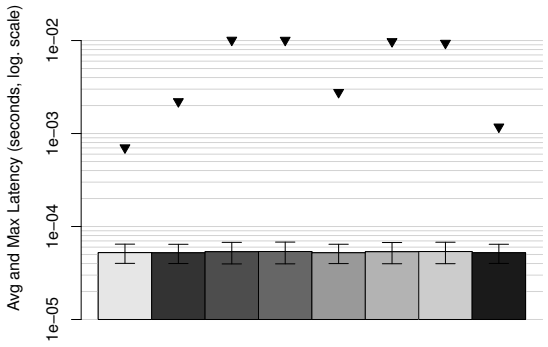

Figure 8. Memory test effects on average (with error bars indicating SD) and maximum latency for the *ReadX* operation in the *tbench* benchmark with one client connected to a server running on the same host (Y axis on logarithmic scale; color coding legend in fig. 6).

been dropped for testing (hotplug and hotplug+shaking claimers, the latter being even more aggressive towards buffer cache frames – cf. subsection VI-B).

- The single-threaded, CPU-bound POV-Ray benchmark is not affected at all.
- IOzone's total runtime seems not to be affected either, but a closer look at only its *reading* benchmarks (fig. 7) reveals that – as expected from the claiming methods' behavior 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.

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 14 from the baseline measurement to the hotplug 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.

VII. 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.

A. Memory Tests

Memtest86+ [4] is an offline memory tester running directly on the hardware of x86-based systems. It performs continuous tests using different test patterns and algorithms such as Moving Inversions and Modulo-X. Memory used by the tester itself is tested in advance to ensure correct operation. Thus, all available physical memory can be tested. In addition to tests under normal operating conditions, stress testing memory outside its specifications is possible by adjusting the memory refresh rate and disabling hardware ECC checks.

Bare-metal memory testing requires downtime of the system. This is usually problematic, however, it reduces side-effects from other running processes. Bare-metal memory testers can be considered complementary to OS-based testers. Their major use is as a hardware-specific tool that is employed if the memory is already suspected of being faulty, whereas OS-based testers can serve as permanent monitoring tools.

A Solaris/SPARC-based online memory test is presented by Singh et al. in [14]. This approach uses a simple, kernel-based architecture that sequentially tests memory ranges by applying the fault model of Nair et al. [6]. The frame-scheduler abstraction provided is rather minimalistic and calculates memory ranges to be tested based on the configured allocation size and the test-iteration number.

Frames with defects are excluded from further use and processes that accessed a frame containing an unrecoverable error are killed. Experimental results are obtained by installing known faulty memory modules into a server and running two different workloads. The detected faults differed between the experiments. Only about one third of the detected errors overlapped – a result that correlates with findings of Schroeder et al. [1], which state that system utilization has a strong influence on the number of memory errors detected by the ECC hardware. Singh's results seem to support this.

At first glance, this system resembles our RAMpage system. However, the pure kernel-mode implementation reduces the flexibility, e.g. in selecting appropriate test algorithms, significantly. In addition, running on a well-defined SPARC platform eliminates many of the problems we had to face with off-the-shelf x86 systems. Singh et al. did not publish results of a performance evaluation, which we consider essential.

Effo GPL is a user-space memory tester for Linux that implements several of the fault models available in Memtest86+. It acquires frames in a straight-forward way by allocating all obtainable memory and 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 [15].

Similar to a bare-metal tester, Effo monopolizes system memory. Since the tester cannot directly acquire specific page frames from user space, it can only do so by allocating many frames and hope that the frame to be tested is included. This monopolizing of resources is a serious shortcoming for a memory test that should not severely degrade the system's performance. Additionally, this approach cannot

predict which frames are to be allocated.

B. Fault Management

Solaris Predictive Self Healing [16] is a framework to improve system reliability using a *fault manager* to interpret signaled hardware errors. If a faulty component is detected, a fault manager tries to offline that component. The system consists of components to start, stop, and restart services [17], to enrich fault logs, and to provide fault prediction in cases where a component's failure is known to be frequently followed by others. 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 [18] is used to handle hardware-related errors, mostly restricted to handling ECC memory errors. It provides an abstraction layer representing the physical layout of memory in modules, down to the 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, and currently, there are drivers for both x86 and PPC architectures. 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. 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.

C. Hardware-based approaches

Dell discusses shortcomings of SECDED (Single Error Correction, Double Error Detection) ECC [19]. 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 ([20], [21], [10] and [19]). It expands the granularity of memory accesses, protecting against the failure of a complete memory chip. Sun UltraSPARC-T1/T2 [22] and AMD Opteron [23] systems implement this strategy by accessing two memory modules simultaneously. While Chipkill is a useful extension to general ECC-based protection mechanism, it is only employed in expensive high-end server systems.

The AMD K8 architecture introduced memory scrubbing [24]. 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 [25] 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".

VIII. Discussion and Future Work

Implementing an online memory tester was a more complex task than originally envisioned. While our general approach is convincingly simple, many details stood in the way of obtaining optimal results.

The most complex task was to understand and partially reverse-engineer the complex and convoluted Linux memory management system. This exhibited one of the areas of Linux that are in dire need of improvement. If the Linux memory management subsystem would provide documented, useful, and invariable APIs, a lot of reverse engineering would have been unnecessary. In addition, our task was aggravated by the idiosyncrasies of the underlying x86 CPU and VM architecture. Many experiments were required until a reliable method of claiming memory from the kernel was procured; by improving the reliability of the system, we had to abandon the idea of testing all available physical memory. So, in some respect, our online memory test is a best-effort approach that nevertheless is able to improve the reliability of a commodity Linux-based x86-64 system. It would be interesting to see how complex an implementation of RAMpage on different Unix-like systems, e.g., BSD, Solaris, or MacOS X, is.

Considering the amount of obstacles to overcome, we think that the achieved results – a high percentage of testable memory, acceptable test times for the complete testable memory range while requiring a comparably low overhead in compute time – are quite remarkable. In order to prepare the memory testing system for real-world use, however, our prototype still has to undergo extensive testing, especially with a larger set of benchmarks.

Additional improvements are closely tied to Linux kernel internals. For example, a tighter integration with the memory management would allow to test an even higher percentage of the overall physical memory. The complexity of Linux memory management [26] and the high rate of changes, however, make this seem a daunting task. In addition, an integration with a Linux-based fault management framework would be worthwhile. However, such frameworks are still in rather early stages of development, especially compared to the functionality provided by Solaris.

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

The implemented approach is unable to avoid errors that show up when a program accesses a page frame before it has been tested. This also implies that all program execution in-between a successful check of a page frame and a subsequently found error on that same page frame may be affected by that error. Thus, the current solution only reduces the probability of an application using a defective memory cell, but cannot avoid the situation completely.

A more invasive change to the Linux kernel that would improve both reliability and memory coverage could test a page frame *before* it is given out to any process or kernel driver *for the first time*. In case a frame is detected as containing a fault, the kernel would then have to choose a replacement frame. Obviously, this approach would introduce a significant latency in page allocations; future experiments will show if a system implementing such a pre-allocation test-policy is able to perform sufficiently well, for example by caching/pooling tested frames until they are needed.

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 and, considering the inherent limitations of these systems, to embedded systems running Linux.

Acknowledgments

We thank Ramin Yahyapour and Jörg Gehrke from the IT & Medien Centrum (ITMC) Dortmund for supplying us with known defective DRAM modules and x86-64 server hardware. We also thank Volkmar Sieh for his support regarding FAUmachine.

This work is supported by the German Research Foundation (DFG) priority program SPP 1500 under grants no. MA943/10-1 and SP968/5-1.

References

[1] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," in *SIGMETRICS/Performance*. ACM, 2009.

[2] B. Panzer-Steindel, "Data integrity," CERN, Geneve, Switzerland, Tech. Rep., 2007.

[3] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs," in *Proc. of EuroSys '11*. ACM Press, Apr. 2011, pp. 343–356.

[4] S. Demeulemeester. (2011) Memtest86+. [Online]. Available: http://www.memtest.org/

[5] IBM, "IBM Chipkill Whitepaper," http://www.ece.umd.edu/courses/enee759h.S2003/references/chipkill_white_paper.pdf.

[6] R. Nair, S. Thatte, and J. Abraham, "Efficient algorithms for testing semiconductor random-access memories," *IEEE Trans. on Computers*, vol. C-27, no. 6, pp. 572 –576, Jun. 1978.

[7] J. P. Hayes, "Detection of pattern-sensitive faults in random-access memories," *IEEE Trans. Comput.*, vol. 24, no. 2, pp. 150–157, 1975.

[8] M.-F. Chang, W. K. Fuchs, and J. H. Patel, "Diagnosis and repair of memory with coupling faults," *IEEE Trans. Comput.*, vol. 38, pp. 493–500, Apr. 1989.

[9] V. P. Srini, "Fault location in a semiconductor random-access memory unit," *IEEE Trans. Comput.*, vol. 27, pp. 349–358, Apr. 1978.

[10] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," *ACM SIGARCH Comput. Archit. News*, vol. 38, pp. 397–408, Mar. 2010.

[11] AMD, Inc., "AMD I/O Virtualization Technology (IOMMU) Specification revision 1.26. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf."

[12] J. Bonwick, "The slab allocator: An object-caching kernel memory allocator," in *Proc. of the USENIX 1994 Tech. Conf., Vol. 1*. Berkeley, CA, USA: USENIX, 1994, pp. 6–17.

[13] S. Potyra, V. Sieh, and M. D. Cin, "Evaluating fault-tolerant system designs using FAUmachine," in *EFTS '07: Proceedings of the 2007 Workshop on Engineering fault tolerant systems*. New York, NY, USA: ACM, 2007, p. 9.

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

[15] C. Cazabon. (2009) memtester website. [Online]. Available: http://pyropus.ca/software/memtester/

[16] Solaris 10 what's new: Predictive self-healing. [Online]. Available: http://docs.sun.com/app/docs/doc/817-0547/esqej

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

[18] D. Thompson, "Linux kernel documentation: EDAC – error detection and correction," [Linux 2.6.35]/Documentation/edac.txt, Aug. 2010.

[19] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," in *IBM Whitepaper*, 1997.

[20] C.-L. Chen, "Symbol level error correction codes which protect against memory chip and bus line failures," US Patent 7093183, Feb. 2001. [Online]. Available: http://www.freepatentsonline.com/7093183.html

[21] S. P. Olarig, "Technique for implementing chipkill in a memory system," United States Patent 7096407, Feb. 2003. [Online]. Available: http://www.freepatentsonline.com/7096407.html

[22] Sun, "OpenSPARC T2 system-on-chip (SOC) microarchitecture specification," May 2008.

[23] AMD, "BIOS and kernel developer's guide for AMD NPT family 0fh processors," http://support.amd.com/us/Processor_TechDocs/32559.pdf, Jul. 2007.

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

[25] Sun Microsystems, "memscrub.c source code, http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/i86pc/os/memscrub.c."

[26] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.