Bad Memory Blocks Exclusion in Linux Operating System

Tomasz Surmacz[1] and Bartosz Zawistowski[2]

[1] Institute of Computers, Control and Robotics,
Wroclaw University of Technology
tomasz.surmacz@pwr.wroc.pl
[2] MSc student at Wroclaw University of Technology
zawistowski.bartosz@gmail.com

Abstract. Memory failures are quite common in todays technology. When they occur, the whole memory bank has to be replaced, even if only few bytes of memory are faulty. With increasing sizes of memory chips the urge not to waste these 'not quite properly working' pieces of equipment becomes bigger and bigger. Operating systems such as Linux already provide mechanisms for memory management which could be utilized to avoid allocating bad memory blocks which have been identified earlier, allowing for a failure-free software operation despite hardware problems. The paper describes problems of detecting memory failures and OS mechanisms that can be used for bad block exclusion. It proposes modifications to Linux kernel allowing a software solution to hardware failures.

1 Introduction

Soft memory errors may be caused by electromagnetic noise and greatly depend on the working environment and the appropriate shielding of the computer system. Cosmic rays are also an important factor, especially in systems with large amounts of system memory or working in vulnerable environments. An estimated error rate of errors caused by cosmic rays is 1 soft error per month per 256 MB of data at sea level [13] (and increasing with height, as the shielding provided by the atmosphere decreases).

Hard errors are defects that are persistent even after rebooting the system. As some studies show [11] not all hardware errors lead to system failures, as some of them may either get cancelled by further data overwriting, or cause silent data corruption. These are the most dangerous to data integrity, as they may remain undetected until a much later time while the system is still running and further data corruption takes place. Also, errors undetected in one of the subsystems may lead to failures appearing in another subsystem [6].

As technology advances, the density of IC elements increases, which allows designing more complex circuits, but also makes harder to produce hardware

W. Zamojski et al. (Eds.): Dependable Computer Systems, AISC 97, pp. 219–231.
springerlink.com © Springer-Verlag Berlin Heidelberg 2011

that is defect-free. Some techniques of dealing with partially defective memory chips have been described in [1] – these include creating mappings for bad bits and storing them in error-free and reliable CMOS, or adding some redundant blocks and permanently blocking the faulty ones at the testing stage. Additional constrains on energy consumption lead to further reduction in currents and charges needed to store a memory cell state. This increases susceptibility of memory cells to random bit flipping from thermal or radiation noise and has to be dealt with at micro- or nanoscale level with appropriate error correction techniques using ECC [1, 10]. These methods however deal with the problem on a circuit design level, trying to provide a view of a reliable and error-free memory chip when seen from the outside. Non-ideal operational environment, overheating or extended exposure to radiation may still cause memory degradation to such an extent, that these techniques fail and some errors start being seen outside. If that happens, the faulty memory chip (or the whole bank) has to be replaced. In Linux, as well as other Unix-like systems, memory management is done through a paging system with the help of a Memory Management Unit (MMU). This allows for a fine-grain exclusion od defective memory regions with a page-size resolution by a Linux kernel with modifications described in this chapter. A system modified this way may be safely run with such partially-faulty memory banks by excluding the faulty regions from system usage.

The rest of this chapter is organized as follows: In sections 2 and 3 we provide short summary of memory errors and testing techniques, in section 4 we discuss memory management in Linux. Our methods of faulty memory exclusion are presented and discussed in sections 5 through 7. In section 8 we summarize the results.

2 Memory Faults and Testing Methods

Memory faults may be combinational or sequential in their nature. Furthermore, these may be either transient or permanent. Combinational faults may be divided in three main categories: stuck-at-0, stuck-at-1 and bridging of two or more lines. They do not depend on the sequence of changes, so generating appropriate set of tests to perform exhaustive testing is easy. In memories, stuck-at faults may either appear in the cells themselves or in the controlling circuitry, e.g. the addressing multiplexers, where they are considered a separate fault category – *address decoder faults*. To test memory cells against the stuck-at errors it is enough to write them with all-ones and all-zeroes patterns and test the resulting values. Bridging of lines and address decoder types of errors require various testing patterns of interleaving zeroes and ones to be written and read from the memory but is still quite straightforward. Some other faults, such as a broken connection between transistors forming a gate or a broken connection between the logic gates can also be modelled and tested as a stuck-at error.

Sequential errors are harder to detect as they manifest themselves only when particular transitions from one state to another take place. As computer memory contains input and output latches, the general sequential fault model has to be applied when constructing test sets.

Functional fault models for memories usually classify faults as *static* (or *simple*) [7], where at most one read or write operation is needed to trigger the fault, and *dynamic*, where a particular sequence of operations is needed for the fault to occur. These can be loosely mapped to combinational and sequential faults, but not as a rule. Furthermore, complex faults may be divided in *linked faults* [7], where some *fault masking* can occur, *single-port* or *multi-port*, and *single-cell* or *multi-cell* (*coupling faults*).

Memory modules without parity checking – known also as *non-parity* modules – are still most common in computer usage. For every physical bit there is exactly one corresponding data bit and no overhead exists. Memory with *parity checking* appends one *parity-bit* to verify correctness of stored data and *non-maskable interrupt* may be triggered that instructs processor to hang up in case of error, so that further data lost can be prevented.

The main disadvantage of parity checking is the lack of error correction mechanisms. It is only possible to check if a memory module is faulty without any opportunity to correct the result of the memory operation, so that further failure-free machine usage could be continued. More sophisticated method is described as ECC [4] (*Error Correction Code*). One of the biggest advantages of ECC is the way the correcting process works – it happens on the fly when errors are being detected. In addition, its implementation is simple and uses only $\log_n N + 1$ control bits for N-bit data input. The cost of applying ECC method is bigger for $N < 32$ and is profitable only for $N > 32$.

The parity generator is also used in ECC but there are several parity bits for data bits. The Hamming distance for coding words is 3 for SEC ECC (*Single-bit Error Correction Code*) so it makes possible to detect and correct only 1-bit errors. A more advanced method – SECDED ECC (*Single-bit Error Correction, Double-bit Error Detection*) allows to correct 1-bit errors too, but it can detect 2-bit errors. It this case, the Hamming distance is 4.

ECC method compared to typical parity checking introduces about 2% speed reduction at average [9].

3 Memory Testing Using Memtest86

Memory faults detection is a complex task, and many testing algorithms have been described in literature. Memory testing can be done either by implementing these algorithms (chosen as best fitting for a particular task) or by using some ready-made software, like Memtest86 [3].

Memtest86 uses *moving inversions* algorithm [2] with several modifications to verify memory modules correctness. It is able to detect two types of faults:

- SAF (*stuck-at faults*),
- AF (*address decoder faults*).

Physical or electrical damages falling in stuck-at category can be easily detected by simple testing algorithms. Bridging, in turn, takes much more tests to be detected, as two or more bits are in the same state independently of written values. Other failures, like faulty addressing lines or a memory that is not present in computer system, are usually easy to detect.

Memtest86 must be run as a standalone program with direct access to memory which may not be obscured by the underlying operating system. Modern Linux distributions include Memtest86 in their bootloader configuration, so when the computer system is started there is a choice to run either the Linux OS or the Memtest86 program (instead of the operating system). When run, the Memtest86 program first builds the memory map by obtaining information from BIOS and analyzing the data provided by ACPI (to skip some reserved locations). It then enters a loop executing a predefined set of tests. If any faulty memory locations are found, they are reported on-screen in a standardized manner, such as:

32-bit address	32-bit mask
`0x03e06e90`	`0xfffffffc`

The address field shows the physical memory address of the beginning of the damaged area and the mask describes which bits are faulty (values set to "1" represent faulty bits). The information given is mostly meant for locating faulty memory chips for the purpose of replacing them. However, the accuracy of identifying addresses and ranges of the faulty areas allows us to use it later to lock the faulty memory pages by kernel and exclude them from the system usage.

4 Memory Management in Linux

Memory in the Linux operating system is organised as follows:

- physical memory is divided in fixed-size pages;
- memory requested by applications is allocated in multiples of page size;
- Linux OS implements a demand-driven memory system [12];
- memory management is supported and supplemented by MMU – if the requested page is not present, it generates page faults and transfers control to the CPU to handle this situation,
- virtual memory provided by the system may be continuous, even though the physical memory does not have to be.

Paging is the default memory management scheme in Linux with typical page size of 4 KB (although Intel processors [8] allow to use 4 MB pages in 32-bit

mode or 2 MB pages in PAE mode). Linux treats pages as the basic unit of memory management. Despite the fact that the smallest addressable unit in processors' word is byte, MMU typically deals with pages. Kernel sources provide several helpful macros that ease the usage of paging in Linux kernel:

- PAGE_SHIFT – determines offset on page (it equals to 12 on x86 machines),
- PAGE_SIZE – defines page size (typically 2^{12} bytes, i.e. 4 KB),
- PAGE_MASK – allows masking some of the offset bits,
- PAGE_PER_PTE, PTRS_PER_PMD, PTRS_PER_PGD – number of entries in Page Table, Middle-level Page Table and Table Directory.

Linux kernel keeps track of every page in order to know its state: what kind of data it contains (kernel code, data structures used by kernel or memory used by user applications) and in how many places it is referenced. The page descriptor is represented by a structure described as follows:

```
struct page {
    /* atomic flags, updated asynchronously */
    page_flags_t flags;
    /* usage counter */
    atomic_t _count;
    /* list of pages, e.g. active_list protected by zone->lru_lock */
    struct list_head lru;
    /* Other fields */
    ...
};
```

There are two fields we should pay attention to:

- `_count` – the usage counter; 0 means that the page is not used, values greater than 0 mean that the page is used by some user processes or by the kernel;
- `flags` – 32-bit or 64-bit number (depending on the kernel version) that describes the status of a page.

All descriptors are stored in `mem_map` table. As the system boots up, all the memory is initially assigned to kernel and one of its main tasks before starting the `init` process and going to multiuser mode is setting up the paging system. This is handled in the `mem_init()` function – it clears unnecessary `PG_reserved` flag in all the pages that are to be returned to the system pool and calculates the total number of pages in the system. Next, it sets appropriate `_count` values for each memory block. Finally, for every page, the `__free_page()` function is called to check if the block is not reserved and if not – decrement the `_count` field, so that the page is returned to the general memory pool and can be used later also by applications running in user-space.

5 Marking Bad Blocks by Linux Kernel

Maintaining proper system behaviour in presence of memory errors requires isolating bad memory blocks and excluding them from further usage. Thanks to the paging system it can be done with granularity of a single page and using some of the existing mechanisms.

The software solution can be basically achieved by two different approaches:

1. A kernel module that detects faulty RAM areas on-the-fly and excludes them from further usage while the system is running.
2. Detection and processing of faulty RAM areas outside of the operating system by an independent testing software and passing appropriate parameters to the operating system.

The first approach requires the whole testing procedure to be known a priori [5]. There are also other problems related to this method:

- every memory access would have to be tested (or checked before the actual operation) in order to eliminate write to a possibly corrupted memory area. This would obviously lead to a significant system slow-down, unless done in hardware,
- testing procedures have to be coded in assembly language in order to eliminate false results while accessing kernel procedures,
- Linux kernel uses as much memory as possible (usually the last 1GB chunk) to provide caching.

The second approach requires stopping the system (i.e. bringing it to a scheduled downtime) in order to check memory for failures, but is universal and can be applied to different computer architectures (testing only relies on other software used in this procedure). Also, testing performed in this mode can be exhaustive and may take as much time as needed to fully and thoroughly test the memory before proceeding with normal system operation.

Availability of Linux kernel source code gives possibility to apply additional fixes to source code wherever they are needed. As it is an open source operating system, we are able to download kernel sources and extend the current memory management subsystem. The modified OS is then able to run on a partially faulty memory hardware as long as some steps are taken beforehand. Memory faults have to be identified first (they can span across several kilobytes of continuous memory address space), but disabling the broken addresses permits the whole system to run stable and without any data corruption.

The following steps have to be taken manually in order to exclude bad blocks from user space usage:

- Save (or write down) all faulty regions from *memtest86* output,
- pass them to the Linux kernel during the booting process,

Disabling faulty addresses inside a kernel is then performed in two steps:

1. Normalize the given addresses with their masks to get a pool of pages that should be marked as locked,
2. exclude them from memory pool by marking the pages as used by the kernel with a special flag, so they will not be reclaimed later.

Parameters passed to grub or LILO are in fact interpreted in the monolithic part of the kernel that starts as the first part of the booting process, so any extensions to parameters syntax have to be actually done in kernel sources, not in grub. If some other bootstrap loader is used (other than grub or LILO), it may be problematic to pass any parameters to kernel at all. In such case the only possibility is to recompile the kernel with the predefined code for exclusion of some predefined memory areas.

Linux kernels (and also the boot loaders such as grub or LILO) impose limits on the size of the parameters that may be passed through them and that limit is set to 255 characters. If the memory faults are numerous, but sparse, some of the information gathered from the *memtest86* output has to be aggregated. In extreme cases, when the faulty block list would still exceed the parameter limits, this may lead to a need of covering also a range of properly working memory addresses just in order to fit in the parameters length limit.

Due to the fact that *address/mask* pairs describe some memory ranges (several faulty regions can be described as faulty by this method), every pair has to be normalised first in order to find addresses of all affected memory pages. This is done while the kernel boots up. By applying two simple bitwise operations it is possible to get a page-aligned starting addresses of the faulty memory block:

```
mod_mask |= ~PAGE_MASK;
mod_addr &= mod_mask;
```

After this normalisation we are able to obtain all other addresses belonging to the faulty range by iterating over all other addresses using the same mask. If it turns out that addresses do not belong to the same memory page – the next one has to be taken into account and be marked as bad too. The modified code executed by the kernel when the memory system initialization is taking place is shown in fig. 1.

While the kernel is booting up, it calculates the number of pages by obtaining the size of memory installed in computer system. When additional information is passed to kernel it marks pages with particular flags so that in further initialization they are used in special manner. In order to exclude faulty blocks from usage, the reference counter cannot be decremented to 0 (as it applies to all pages used by kernel), as the page could be later reclaimed for some other use. It is possible however to extend the paging system by adding some custom-defined flags to the ones already defined

```
unsigned long a; /* address */
unsigned long m; /* mask */
unsigned long oa = a;
unsigned long r = (a | m) + 1;
if(!r) {
    return 0; /* overflow? */
}
r = (r & ~m) | (a & m);
if(r < oa) {
    return 0;  /* mark the whole area as bad */
}
return r;      /* next address found */
```

Fig. 1 Modified memory initialization in `mem_init()` kernel function

in `include/linux/page-flags.h`. We have introduced an extra flag called `PG_memlocked`, as show in fig. 2.

```
enum pageflags {
  PG_locked,
  PG_error,
  PG_referenced,
  PG_uptodate,
  PG_dirty,
...
  PG_memlocked /* new page state */
};
```

Fig. 2 Extra flag defined in include/linux/page-flags.h

This flag is used to mark all faulty pages as the system boots up and prevents the `__free_page()` function run at the end of the `mem_init()` procedure from returning the page to the general memory pool. As the locked-out page is not going to be used by any process, it would be a very likely candidate for swapping out. But swapping for a locked-out page would effectively mean that it gets reused and allocated to some other process, while the unreferenced, but locked-out page is permanently moved to the swap space. So swapping for locked-out pages has to be prevented and the simplest method it to keep them as kernel-allocated pages (by setting the `PG_locked` flag in addition to `PG_memlocked`), as kernel pages are never swapped-out.

Address/mask pairs can be passed during booting process from boot loader like *grub* or can be compiled into kernel by checking *Built-in kernel command line* option during kernel configuration for custom build. Both methods may require increasing the value of `COMMAND_LINE_SIZE` defined in `include/asm-generic/setup.h` file.

6 Testing Methodology

Testing the operating system stability during typical system usage is problematic as several processes are running concurrently to provide different services needed for normal system operation. The set of running processes should be minimised to facilitate easier isolation and replication of problems. There are two basic ways of detecting the system being unstable due to the memory corruption:

- Kernel crashes while obtaining more memory, e.g. while loading a device driver,
- Application that allocates memory and writes a predefined pattern does not get the same data during the read operation.

The first case results in system panic that leaves traces in appropriate log files, which can be checked by executing `cat /var/log/messages /var/log/kern.log | grep panic`. The second case cannot be easily detected under normal system operation, but requires some testing software to be run. We can distinguish two kernel failures:

- Hard panic ("*Aieee!*") – only a system reboot is possible to recover from failure,
- Soft panic ("*Oops*") – system can be still used but a particular operation did not succeed.

Hard panics happen usually in interrupt handling routines of the drivers, basically because of a null pointer dereference. Later, the device driver cannot handle incoming interrupts and causes the kernel panic. Soft panics happen outside of interrupt handling code and allow the operating system to continue, but without the crashed driver. Hard panics are signalled by blinking keyboard LEDs and freezed screen output. No input can be handled by the system and only a hard reset is possible. The stack trace is rarely logged to `/var/log/messages` file and only the console screen dump can be useful to get some information about the failure. Soft panics may provide more useful information, but some effort is required beforehand – "`Kernel Hacking -> Detect Soft Lockups`" option has to be enabled *a priori* in order to get all debug messages during system crash. Writing crash info to a file may still not be possible (due to a crashed driver or just because of inability to "sync", i.e. commit the memory-buffered write operations to disk), so it is also beneficial to set-up a serial console. This can be done by enabling "`Device Drivers -> Character devices -> Serial drivers -> Console on 8250/16550`" and appending `console=ttyS0,115200` to *grub* or *LILO*. Lastly, to log the stack trace of drivers working in non-interrupt-driven mode, "early printk" has to be enabled by selecting "`Kernel Hacking -> Early printk`" in `.config` file stored in root of Linux kernel source directory.

To test user-space memory for possible corruption within a running system we run a test application that allocates as much memory as possible and performs multiple write/read tests. There is however a limit on the amount of memory that can be loaded by an application without getting a `SIGKILL` signal. The smallest chunk that can be allocated is the page size defined as `PAGE_SIZE` constant. For this reason there is no need to call `malloc()` with smaller values than 4 KB for typical 32-bit architecture. To make the solution universal we call `getpagesize()` system function, which returns the page size of the operating system. Similar results can be achieved by `getconf PAGESIZE` command in shell command prompt or `sysconf(_SC_PAGESIZE)` function. We use the first approach. Figure 3 demonstrates how much memory can be allocated by a program executed in user-space.

```
#include <unistd.h>
#include <stdlib.h>

int page_size = getpagesize();

int i = 1 ;
void *p;
for (;;) {
  p = malloc(page_size * i);
  if (p==NULL)
    break ;
  free(ptr);
  ++i;
}
/* (i - 1) now stores the number of pages */
return (i-1);
```

Fig. 3 Code that calculates the number of pages that can be allocated by a user process

After calculating the maximum number of pages that can be allocated, each fixed size chunk has to be tested with write/read patterns to find possible inconsistencies.

The overall testing process can be described as follows:

1. Check all necessary logging options in kernel configuration before compiling the kernel.
2. Modify bootolader configuration file to enable serial console.
3. Boot up the system with new kernel in single-user mode.
4. Disable swaping by calling `swapoff` or by editing `/etc/fstab` file and modifying the entry related to swap partition so that it will not be mounted.

5. Calculate the number of pages that can be allocated by a user-space program.
6. For each test pattern check allocated memory by `assert()` macro.
7. Load all the drivers as modules and check for hard kernel panics.
8. Check for "panic" entries in syslogd files.

7 Limitations and Further Perspective

Presented solution does not require any additional hardware and is only based on a software kernel extension. Typical booting process for an x86/i86 architecture personal computer is a quite clearly described process. At the very early stage, the processor operates in real mode and only 640KB of memory can be used. Kernel loading can be divided in two stages:

1. A smaller piece of code that is loaded somewhere below the first 640 kilobytes of memory, which is often called as bootstrap.
2. A bigger part loaded above 1 MB of memory where kernel operates in protected mode.

In addition to that, some kernel modules may be loaded at later time, triggered by configuration options or a hardware-detection code that gets executed during the system startup. The critical part of the booting process happens however in the mentioned two early stages, as both of them need to be loaded in memory parts that are not faulty. The kernel code is loaded unconditionally without a choice of preferred memory addresses to use, and if some pages happen to be faulty, the results may be unpredictable. High level initialization takes place in `start_kernel()` procedure where memory management is being set up and then faulty regions may be mapped to pages. Kernel modules pose no problem, as they are loaded after the memory system is initialized and the faulty pages are already mapped-out.

If the memory chips are damaged in such a way that the early-stage kernel code is loaded into faulty pages, the only solution may be to change the system memory by manually swapping the physical memory banks (i.e. placing them in different memory slots so thet their physical addresses are arranged differently) ore use some fault-free banks for the physical addresses used by the bootloader. In typical systems it is the last gigabyte of physical memory.

Another issue may appear in systems where page size is bigger than typical 4 KB. If we consider a 4 MB page and huge amount of faulty addresses spread out on not consecutive areas, then even small and sparse errors may cause exclusion of large address ranges from further use. Such systems are however in the experimental stage of deployment, and until they are widely used, we may investigate other solutions that will access and exclude memory blocks on a sub-page level.

The current solution has been applied and tested on early versions of 2.6 kernel series. Several data structures have been changed since then – for

instance, the `page_flags_t` type has been changed to `unsigned long` and some other additional memory features have been added to the latest kernel versions.

Current versions of the memory testing software do not support dumping the test output to a filesystem installed on a device (i.e. the hard drive or a USB memory stick), so it may be quite difficult and troublesome to save all the addresses that are produced on screen by the testing procedure. Memtest86 operates in real mode so other extensions may not be easy to implement, it is however one of the concerns for further development, as it would greatly simplify the automatization of the whole process.

Buffer size for command line parameters is around 256B so it may be impossible to pass all faulty addresses to Linux kernel during boot. In such case, the kernel has to be modified to include a `mem_init()` function extension with a predefined list of pages to exclude. It can be done by hardcoding a table of `unsigend long` *address/mask* pairs that will be used in addition of the parameters passed though the bootloader, This approach requires the kernel to be recompiled every time the pool of faulty regions changes, which may be tricky on a system that must run with these errors, so it must either be done by a kernel compilation performed on a different machine, or by a temporary addition of new faulty memory addresses through the bootloader parameters. This way that the system may run stable while the kernel is recompiled with an updated map of faulty memory locations. The preparation of such a system setup may be a tricky one, even though the proposed solution allows for a safe and complete exclusion of faulty memory blocks.

8 Conclusions

Linux kernel modifications described in this chapter allow fault-free operation of the Linux OS on a hardware where memory banks are partially faulty but the faulty addresses may be identified by some external testing programs. Faulty memory regions are excluded from system usage with page-size granularity by locking them as unswappable kernel-used memory, without actually accessing them for any purpose.

The described modifications have been implemented and tested in 2.6 series of kernels, up to version 2.6.6, and the implementation effort is now directed to porting them to current versions, for both the 32- and 64-bit systems.

References

[1] Biswas, S., Metodi, T., Chong, F., Kastner, R.: A pageable, defect-tolerant nanoscale memory system. In: IEEE International Symposium on Nanoscale Architectures, NANOSARCH 2007, pp. 85–92 (2007), doi:10.1109/NANOARCH.2007.4400862

[2] De Jonge, J.H., Smeulders, A.J.: Moving Inversions Test Pattern is Thorough, Yet Speedy. International Computer Design (1976)

[3] Demeulemeester, S.: Memtest86, an Advanced Memory Diagnostic Tool (2010), http://www.memtest.org
[4] Elkind, S., Siewiorek, D.: Reliability and performance of error-correcting memory and register arrays. IEEE Transactions on Computers C-29(10), 920–927 (1980), doi:10.1109/TC.1980.1675475
[5] Elm, C., Klein, M., Tavangarian, D.: Automatic On-line Memory Tests in Workstations. In: Records of the IEEE International Workshop on Memory Technology, Design and Testing (1994)
[6] Gu, W., Kalbarczyk, Z., Ravishankar, Iyer, K., Yang, Z.: Characterization of Linux kernel behavior under errors. In: Proceedings of 2003 International Conference on Dependable Systems and Networks, pp. 459–468 (2003), doi:10.1109/DSN.2003.1209956
[7] Hamdioui, S.: Testing static random access memories: defects, fault models, and test pattern. Kluwer Academic Publishers, Dordrecht (2004)
[8] Intel Corporation IA-32 Intel Architecture Software Developer's Manual. In: System Programming Guide, vol. 3 (2009), http://www.intel.com/products/processor/manuals/
[9] Kozierok, C.: The PC Guide – Memory Errors, Detection and Correction (2001)
[10] Maestro, J., Reviriego, P.: Selection of the optimal memory configuration in a system affected by soft errors. IEEE Transactions on Device and Materials Reliability 9(3), 403–411 (2009), doi:10.1109/TDMR.2009.2023081
[11] Messer, A., Bernadat, P., Fu, G., Chen, D., Dimitrijevic, Z., Lie, D., Mannaru, D., Riska, A., Milojicic, D.: Susceptibility of commodity systems and software to memory soft errors. IEEE Transactions on Computers 53(12), 1557–1568 (2004), doi:10.1109/TC.2004.119
[12] Silberschats, A., Peterson, J.L., Galvin, P.B.: Operating System Concepts. Addison-Wesley Publishing Company, Inc., Reading (1991)
[13] Swift, W.: Memory errors: roll the dice! IEEE Antennas and Propagation Magazine 38(6), 124–125 (1996), doi:10.1109/MAP.1996.556530