Diplomarbeit

Proactive Memory Error Detection for the Linux Kernel

Jens Neuhalfen

18. Juni 2010

technische universität
dortmund

Fakultät für
Informatik

Institution

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12 für Eingebettete Systeme

Gutachter

Dr. Michael Engel
Prof. Dr. Olaf Spinczyk

Contents

List of Figures

Listings

Chapter 1

Introduction

Errors in computer memory have been the subject of scientifically and economically motivated research for a couple of decades. This research led to various hardware and software based detection and correction schemes for memory errors.
This diploma thesis discusses the question

> "How could a feasible software based memory test for low- to midrange servers be designed?"

by identifying and analyzing its crucial aspects. Additionally, the hereby elaborated aspects will be put into practice by implementing and evaluating a software based memory test for the Linux kernel. Further, this diploma thesis aims to lay the groundwork for research in the area of software based memory testers. Visualising the usage and state of the physical memory can greatly help to understand, what is happening inside memory management systems.
The purpose of this diploma thesis is to minimize the chances of encountering undetected data corruptions in memory. Technically, the purpose is to detect ill-behaving or broken memory modules as early as possible, minimizing the period in which the module corrupts data.
The means to accomplish this is to proactively run a memory tester like *memtest86+* [13] in the background and detect errors before they cause harm to data. In case the detector finds ill-behaving memory, a fault management process that removes the memory from usage by the Linux kernel will be

initiated. Additionally, a theoretical discussion of a more elaborated fault management process will be included.

For the best code and knowledge reuse, and in order to support the implementation of a Linux fault management architecture, the implementation will support and build on the Linux kernel and its failure handlers. This chapter motivates the goals of the thesis, and provides context and background for the motivation.

This thesis is based on employing existing hardware, software and empirical research. A distinction between this thesis and related work will be presented. The following sections are the introduction to the analytical and evaluational goals of this thesis.

1.1 Implications of Defective Memory

Today's Age of Information Technology has an increasing impact on lifestyle and business. As a result the quality and reliability of software and hardware systems is of particular importance [47]. Companies and researchers alike take significant efforts to improve software quality by various means. Examples of these efforts can be found in [41, 91, 90, 75] and many other resources.

1.1.1 The Effects of Hardware Reliability on Software Reliability

Software makes assumptions about the environment it runs in. E.g., layered software generally relies upon correctly working layers below. The lower layers provide services by exporting interfaces that are used by higher layers or client programs. In most cases an interface consists of multiple elements including the binary interface, machine readable descriptions of the interface, and human readable documentation. Together they form what often is called a *contract* that describes which services are provided under which pre- and postconditions. Both signers, the client and the service, expect each other to fulfill this contract. A detailed discussion of *design by contract* can be found in [91]. A party that does not abide by the contract is said to be *misbehaving*. The behaviour of client and service can be undefined when either of them misbehaves.

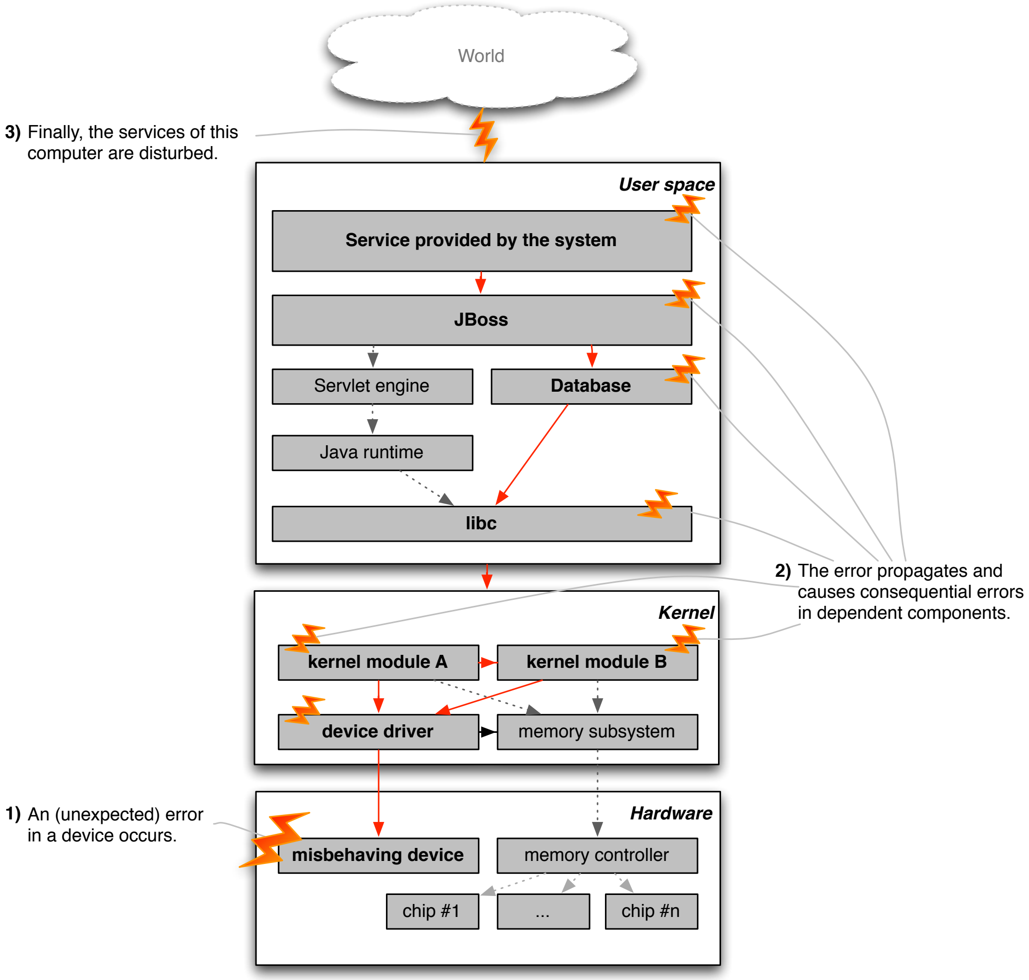

World
3) Finally, the services of this computer are disturbed.
User space
Service provided by the system
JBoss
Servlet engine
Database
Java runtime
libc
2) The error propagates and causes consequential errors in dependent components.
Kernel
kernel module A
kernel module B
device driver
memory subsystem
1) An (unexpected) error in a device occurs.
Hardware
misbehaving device
memory controller
chip #1
...
chip #n

Figure 1.1: *Example of error propagation in a software stack.*
This image shows a simplified typical software stack for a business application. Grey boxes are software components and arrows denote dependencies between components.
At the bottom of the image, an unexpected error happens in a device. The device fails to handle the error properly and passes it to the driver(**1)**, *at the bottom*), leading to a violation of the contract with the driver. This misbehaviour causes consequential errors by the driver and other components (**2)**) that spread up through the software stack and lead to disturbances in the services the computer provides (**3)**, *at the top*).

In other words, the correct behaviour of a running program depends on the correct behaviour of all services that it depends on directly or indirectly. Examples for this are the use of libraries, the Java Virtual Machine, the operating system, or the hardware – as will be discussed in the following.

1.1.1.1 Memory Errors

Hardware errors that cause memory corruption are an especially dangerous kind of error. They can corrupt the data handled by the software and the software itself, potentially affecting every service.

Bit errors in memory are a significant threat to the integrity of data. While a flipped bit in a computer game might only slightly discolour a pixel, it can have severe consequences if the data is the foundation for important decisions.

Often, it is very difficult, maybe even impossible, to determine whether or not a given data set is corrupted, especially if memory errors are not detected the moment they happen. In essence, undetected bit errors violate a very basic axiom of a hypothetically underlying Turing machine: Data on tape can only be changed by the head and does not change due to outside factors.

It is important to distinguish between two forms of *misbehaviour*: a client that uses a service incorrectly, and a service that violates its own contract. But even under the optimistic assumption of a correctly implemented client, a client that correctly uses a service can still be disturbed by a misbehaving service, as shown in figure 1.1. Therefore this diploma thesis considers the hardware to be the lowest layer of the system and only misbehaviours originating in the hardware will be examined. In detail, the software based detection of RAM defects will be discussed. This can be understood as verifying the contract between the operating system and the physical memory. It will be assumed that higher layers work correctly. Thereby, the problem space of misbehaving services is narrowed up to a point where the number of failure scenarios corresponds to the scope of this thesis.

1.2 Identifying Defective Memory

In mid 2009 Google and Bianca Schroeder from the University of Toronto published a study (see [105]) that showed that Google's – albeit commodity grade – server hardware is prone to memory errors: Each year a third of the studied systems suffered at least on correctable memory error (CE). According to their study, a system that had a CE in the past is very likely to have much more CEs or even uncorrectable errors (UE) in the near future.

It is safe to assume, that consumer grade hardware[1] is likely to show worse behaviour. These results emphasize the importance of the early detection of defective memory modules.

1.2.1 Hardware-Based Counter Measures

The dangers of errors in computer memory and caches lead to the development of countermeasures, with the most prominent and most widely implemented of them being ECC (*E*rror-*C*orrecting *C*odes). In this context, ECC stands for memory subsystems equipped with *in situ* detection and correction capabilities of certain error classes. The basic form of ECC is termed SEC-DED (*S*ingle *E*rror Correction, *D*ouble *E*rror *D*etection) and allows the detection of two bit errors and the correction of one bit errors, for details see Levine and Myers [83] and Chen [50].

Various vendors improved ECC for systems that need a greater level of confidence in the correct behavior of memory. The list of ECC-improvements includes IBM (see the whitepaper *Chipkill*[70]), *Extended ECC* (patent entry [67]), Intel *SDDC* (described in an application note [73]) or *Chipspare* from HP. These technologies are mostly used in more expensive server systems. The same is true for Intels *MCA Recovery*, a mechanism that allows operating systems to detect and potentially repair/circumvent hardware errors, for details see the whitepaper [72].

In 2010 Yoon and Erez presented ongoing research about virtualized ECC. By

[1] Consumer grade hardware is commodity hardware targeted at office and home users. The price of these system is often more important than their reliability.

modifying a memory controller by hardware simulation and by extending the operating system, they showed that ECC checksums can be stored in regular memory, as opposed to the special memory used today[119]. They also showed how important data could be selectively protected by more potent detection and correction schemes.

1.2.2 Software Based Counter Measures

Software-based counter measures against defective memory have a great disadvantage compared to hardware-based counter measures. They cannot detect memory errors the moment a client uses the memory because the interaction between memory and client is private to the client.

Most software based counter measures work as memory testers that write different patterns to the memory and verify that the memory works correctly by reading the data. These software memory testers run either on the bare metal or in cooperation with the regular operating system that the computer runs.

Bare metal memory testers like *memtest86+* prevent any other usage of the computer while the tests run[13]. This makes them infeasible for timely detection of defective memory, because a downtime must be scheduled for each run.

At first, the act of writing a memory tester that runs under an operating system seems simple: Create a kernel thread that iterates over every frame and tests each frame. Sun implemented a memory tester prototype with this architecture for Solaris 8, see [109] for details. Although this simple algorithm might work in practice - and Suns implementation did find memory errors - it ignores several important factors that need to be considered.

Listing 1.1 is an example implementation of such a simple tester and is used to highlight several aspects of a memory tester. First of all, there is the question of *frame acquisition*, implemented in $acquire\_frame$. Operating systems manage the available memory, each page frame is either allocated or free. Free memory can be allocated to the memory tester, but memory that is allocated to other entities must be reclaimed before it can be used.

```c
/*
 * Simple scanner that tests all frames for errors
 * and marks bad frames by calling ‘mark_frame_bad‘.
 */
void test_all_frames(size_t max_pfn) {
  // Frames are numbered  [0..max_pfn[
  for (size_t page_frame_number = 0;
      page_frame_number < max_pfn;
      page_frame_number++){
    // Test for missing / already marked as bad frames
    if (! is_page_frame_marked_bad(page_frame_number))
      if (IS_OK( acquire_frame(page_frame_number)))
        // ‘test_frame_for_errors‘ returns true, when the frame showed errors
        if (test_frame_for_errors(page_frame_number))
          mark_frame_bad(page_frame_number);
        else
          release_frame(page_frame_number);
  }
};
```

Listing 1.1: *Example of a simple memory tester*

`test_all_frames` is a show-case implementation of a simplicistic memory tester.

The method `acquire_frame` tries to allocate the specified page-frame for the caller on a best-effort base.

`test_frame_for_errors` tries to find errors by writing and verifying different testpatterns to the page frame.

Depending on the circumstances it can be fairly difficult or even impossible to claim a currently used frame for a memory test. A frame can be acquired at nearly no cost the moment a client releases it to the memory manager, while an actively used frame from the scarce DMA(direct memory access) region would be very expensive to claim. Testing memory from such restricted memory regions can also have a high impact on system performance and stability, when the memory tests starves other parts of the kernel of memory frames.

1.2.2.1 Scheduling which Frames to Test

This leads to the idea of a *Frame Scheduler* that determines which frame should be tested when and how. By iterating over each frame, the implementation sketched in listing 1.1 implements a very straightforward scheduling strategy. More sophisticated schedulers would implement a better weighing of different interests.

1.2.2.2 Scheduling Testing

Testing memory for failures can be a very resource consuming activity. An exhaustive test for an n-bit memory would need to test $O\left(\left(3n^2 + 2n\right) 2^n\right)$ combinations [65], this is obviously infeasible for memory sizes larger than a few bits. Algorithms that detect a restricted set of errors can be written in $O(n)$ [102]. Limited resources like memory bandwidth, CPU cycles or lock contention suggest that even an $O(n)$ implementation would consume significant resources, leading to the observation that the scrubbing process should be scheduled as well.

1.2.3 Conclusion for Low-End Servers

The most important thing about ECC and the other hardware improvements is: they have to be integrated into the system. It goes without proof that most home and office computers and many low-end server systems do not have ECC memory installed and thus are more susceptible to memory errors.
Using a software based memory tester on these low-end systems improves the chance that defective memory can be found earlier. Systems equipped with ECC could also benefit, because a thorough memory test would write many different patterns into the memory, increasing the probability to detect errors that affect more bits than covered by the ECC detection.

1.3 Handling Defective Memory

Actions have to be taken if a defective frame is found. The Linux EDAC project supports frame-to-module mapping for certain chipsets and mainboards, the more actively maintained mcelog package provides a similar set of features, for details on the projects see their respective websites at [4] and [30]. Both projects allow a defective frame to be related to a specific physical memory module. This allows the operator to find and replace the impacted module.
Lastly the question of fault management remains to be solved. Currently no sophisticated and unified fault management architecture for Linux exists. The approach mostly taken is logging and/or halting the system. More sophisticated

automatic responses to error symptoms could greatly increase the reliability of Linux as a platform.

1.3.1 Fault management

Faults must be handled to ensure the stability of a system. Fault management is the process of responding to detected faults[2] . The correct response to a fault heavily depends on the context the fault happens in and the policy set in place for the detected class of error.

In section 1.2.2 the question was posed, what should be done, when an ill-behaving memory frame had been found by the software based memory tester.

The study by Google, mentioned in section 1.2.1 [105] suggests, that the error is likely caused by a hardware defect and not caused by external factors like alpha particles (compare to [115]). The rapid pace of microminiaturisation of chip structures will probably increase the transistors' susceptibility to external influences like temperature and cosmic rays. Both the Deutsche Forschungsgemeinschaft (DFG) and the International Technology Roadmap for Semiconductors (ITRS) see this as a challenge for the near future [66, 106].

Googles study further indicates, that memory modules with an existing error are more likely to have additional errors than modules without known errors. Increasing the efforts to examine all frames of the suspect memory module would be a prudent reaction to errors found in it. Possible long term actions could be the removal of the affected memory module by an operator or a thorough examination of the hardware.

Modern systems like the Intel-*Nehalem EX* family of processors allow the operating system to react on errors detected by the ECC-system, different action can then be taken, depending on whether a correctable or an uncorrectable error has been detected.

[2] According to Mukherjee, user visible errors are manifestations of underlying faults [94].

1.3.1.1 Considerations for defective memory fault handling

If a defective memory frame has been found by the software tester, then no other part of the systems should be affected, since the frame is solely allocated to the software memory tester. Errors detected by ECC are more difficult to handle because errors in memory areas allocated to different subsystems or client applications must be handled with extra care. These errors might affect program flows that are not equipped to react properly to corrupted memory.
Specifying or implementing a fault management system for Linux is beyond the scope of this thesis. A description of fault management in general and how the detection of and reaction to defective memory could be embedded into a fault management implementation rounds off the subject.

1.4 Goals of this thesis

The goal of this thesis is to answer the question stated in the first section:

> “How could a feasible software based memory test for low- to midrange servers be designed?”

This question can not be answered in general, because important aspects of *feasibility* can only be answered for a specific context.
A running memory test consumes resources such as CPU-runtime, free memory and puts stress on the memory bus. A given overhead that is acceptable for systems with high requirements for stability will probably be evaluated differently in environments, where performance is more crucial than timely detection of defective memory.
Hardware differences between systems can also affect the runtime behaviour of a memory test. Aspects that might lead to different results include, beside others, the memory type used, the number of DIMMs used, the memory controller, the number of processor cores, the number and type of NUMA-memory domains, the usage of cache coherence protocols or the memory bandwidth.
Systems with similar hardware and similar requirements regarding performance can still exhibit very different behaviour. Some systems run very memory intensive applications, some systems run CPU intensive applications and other

systems are very I/O-bound. Most systems will probably run a specific mixture of these loads. Performance is a multifaceted metric and includes aspects like *throughput*, *latency* and *scalability*. Each of these facets could be influenced by running a software based memory tester.
If the question of feasibility can not be answered in general, it will be answered *exemplary* for a given system and given workloads.
To make this possible, this thesis will accomplish four different goals:

- Design and implement tools to visualize the usage of the physical memory frames.
- Design and implement a software based memory tester for the Linux kernel and userland, including mechanisms to define different scheduling policies on which memory frame is scheduled to be tested when and how.
- Establish performance tests and evaluate the performance impact of running the implemented memory tester under different scheduling policies.
- Describe the current status of fault handling in Linux, compare it to fault handling as it is implemented in Solaris and propose how the Linux fault handling process could be improved.

1.5 Structure of this thesis

In this chapter the idea behind this diploma thesis has been explained, the usage of hardware- and software based mechanisms to detect faulty memory has been motivated, and the goals of this thesis stated. In chapter 2 the problem will be described in more detail. This includes the introduction of fault models, a description of fault handling, including a vision for a unified fault handling under Linux. Further, the Linux memory management (mm) will be introduced, and aspects especially relevant to this thesis will be discussed in more detail. In the third chapter then relevant work, both hardware- and software based, is presented and discussed. In chapter 4 the design of the implementation is presented, but before that requirements for the implementation are elaborated. Implementation details are discussed in chapter 5, followed by the evaluation in chapter 6. Chapter 7 concludes this thesis with a retrospective of the implementation, and an outlook on further developments.

Chapter 2

Problem Description

This chapter describes the problems highlighted in the first chapter in more depth. Starting with a functional description of how RAM is typically abstracted by the hardware memory subsystem, the origins of memory faults are explained on a level suitable for this context. The requirement to detect faults in memory poses the question, how faults in memory can be detected by a program running on the CPU. Several fault models for memory hardware have been proposed in the last decades, two of them, the PSF-fault model introduced by Hayes [65], and the fault model introduced by Nair *et al.* [95] are introduced. Both models origin in the mid seventies and are the base for many other fault models. These fault models where chosen for two reasons: First, they are both fault models that are the base for many fault models used today. Secondly they show a very different complexity. Testing Hayes' fault model requires a number of tests that is exponential to the number of bits in the memory, and testing Nair's model can be done in linear complexity.

After the theoretical basis for fault detection in RAM is described, the topic of fault *management* is introduced. To ensure the reliability of a system, the detection of faults is only the first step. Once a fault has been detected, or suspected, counter measures can be initiated. Section 2.3 gives a birds eye overview of fault management and describes how a Fault Management Architecture (FMA) can be implemented to handle faults in memory.

The proof of concept implementation stipulated in chapter 1 is targeted for the Linux kernel. Section 2.4 justifies Linux as implementation target, and introduces the relevant aspects of memory management.

The following sections use the word “frame” in advance of the discussion in section 2.4. For now, the explanation of frame is, that Linux manages memory in units of frames. A frame consists of a fixed number of consecutive bytes, and each frame is uniquely numbered by its Page Frame Number (PFN).

2.1 What Causes Faults in RAM

It makes sense to distinguish between *externally* induced errors, and *innate* errors of the physical chip. Externally induced errors are often called *soft errors*, errors that are caused by physical defects are called *hard errors*. The line between hard and soft errors is somewhat blurred, taking for example a chip that behaves well at 30°C and shows errors at 50°C. A major cause for soft errors are neutrons and cosmic rays [98].

Hard errors are physical defects like shorts caused by metalization of memory chips, buses, or the memory controller, particle contamination of the silicon, or voltage differences between the ground level of memory controller and module, and many more [86]. Hardware that suffers from hard errors should be replaced as soon as possible to uphold the reliability of the system.

2.1.1 Reliability

The reliabilty of memory modules is judged by their Failures In Time per billion device hours (FIT)-rate[1]. It is unclear, what can be considered the average FIT rate of computer memory because different research groups deliver different results. To quote from a study done by Google[105]

> We observe much higher error rates than previous work. Li et al cite error rates in the 200–5000 FIT per Mbit range from previous lab studies, and themselves found error rates of $<$ 1 FIT per Mbit. In comparison, we observe mean correctable error rates

[1] see the glossary entry for a definition.

> of 2000–6000 per GB per year, which translate to 25,000–75,000 FIT per Mbit.

Despite all difficulties in distinguishing between hard- and soft errors, and the stark variations in the detected average FIT rate of memory, it can be taken for granted that computer memory *will* show errors. These errors have to be assessed depending on the context and the frequency they happen at. Fault management is discussed in section 2.3.

2.2 Fault Models for RAM

Modern computers are behave according to the *van Neumann model* shown in figure 2.1[97]. The system memory (RAM) plays a central role in this computer model because it stores the program, the source data for and the results of all calculations of the processor. Although modern system architectures extend this model by various layers of caches and backend storage, the general principle still applies.

A fault model of a system describes the correct behaviour and the to-be-expected types of faults of the specified system. Specifying a fault model allows the prediction of the consequences of faults, and the design and verification of fault detection and prevention measures. Fault models can be related to the contracts between a service and its clients described in section 1.1. In addition to the correct behaviour of the system, a fault model also includes possible misbehaviours of the system, it can be said that a fault model supersedes the contract. To draw on the contract of a *malloc* service found in the C library, a fault model for the *malloc* service would describe the correct behaviour of *malloc*, and additionally could include faults like "failure to release memory upon $free$", or "the memory area returned by *malloc* is too small".

The following sections describe memory fault models for RAM and research efforts to detect the described faults.

In 1974 *John P. Hayes* published a paper on *Detection of Pattern-Sensitive Faults in Random-Access Memories* [65]. In it he described a functional model of RAM that abstracts from the physical implementation of RAM. Further, he

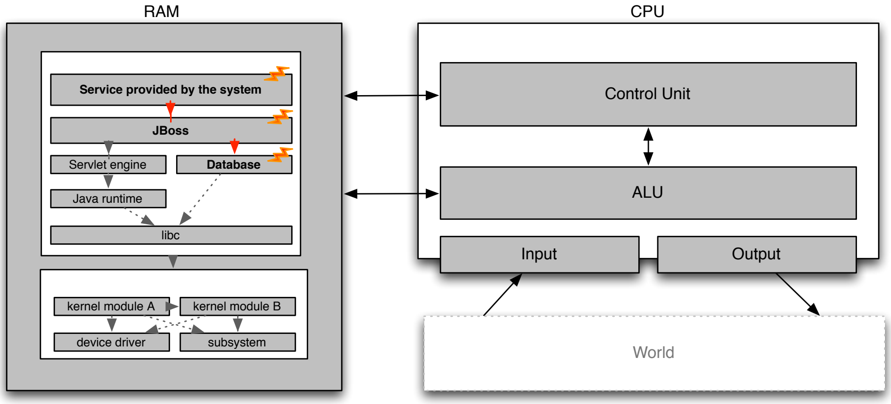


Figure 2.1: *Classical van Neumann architecture.*
This image presents the classical *van Neumann* architecture for computers [97]. The memory (RAM) contains both instructions, and data for the CPU, indicated by the software stack that served as an example in the first chapter.
In this case a hardware defect in the memory causes faults in the database, that then propagate through the software stack. In contrast to the example in figure 1.1, where a faulty device caused an error that started to spread from a specific location, a memory error can affect each and every software component currently in memory. If the operating system employs techniques like memory deduplication or shared libraries, one fault can affect multiple services simultaneously.

proposed a fault model that covers a large range of memory faults based on his logical model of RAM, and algorithms to detect these faults.

In contrast to prior research, his thesis focuses on faults on the *functional level* of random access memory, thus concentrating on the correct functional behaviour and excluding the physical layout, and behaviour of the underlying hardware. Research that reckons physical and layout aspects can be found in [49, 118, 111] and others.

By focusing on the functional level it is possible to test for defective memory without knowing the internal physical layout of the memory, and the whole memory hierarchy. This is an important aspect, because the goal of this thesis is to detect errors on server hardware that runs Linux. It is unrealistic to assume that detailed data about the whole physical memory subsystem is available, as this includes detailed information about internals of the installed processor and memory controller, the mainboard, and the equipped memory modules.

2.2.1 The Logical Structure of Memory

This section summarises the memory model that Hayes described in his publication [65]. Hayes' model of memory is used as base, and referenced throughout this thesis. To improve comparability with the original paper, the notation and phrasing used here stays close to the ones Hayes used.

Hayes defined *memory* as a set of r addressable binary storage cells $M_r := \{C_0, C_1, \ldots, C_{r-1}\}$, where i denotes the address of C_i and where each storage cell C_i can store exactly one bit.

Operations $READ$ and $WRITE$ can be performed on M_r, and each cell can be written to or read from independently of previous $READ$ or $WRITE$ operations: M_r is a *random-access memory*.

A memory of r cells stores r independent bits, leading to a total of $n = 2^r$ distinct memory states. $Y_{0..n-1}$ denotes the possible states a memory M_r can be in. A single state Y_j can be described by a vector $(y_{j,0}, y_{j,1}, \ldots, y_{j,r-1})$, with $y_{j,0..r-1} \in \{0, 1\}$ being the content of C_i.

The operations $READ$ and $WRITE$ 0/ $WRITE$ 1 are associated with the functions W_i, $\bar{W}_i$ and R_i on the states of M_r with

$$W_i(y_{j,0}, y_{j,1}, \ldots, y_{j,\mathbf{i}}, \ldots y_{j,r-1}) = (y_{j,0}, y_{j,1}, \ldots, 1, \ldots y_{j,r-1}) \qquad (2.1)$$

$$\bar{W}_i(y_{j,0}, y_{j,1}, \ldots, y_{j,\mathbf{i}}, \ldots y_{j,r-1}) = (y_{j,0}, y_{j,1}, \ldots, 0, \ldots y_{j,r-1}) \qquad (2.2)$$

$$R_i(Y_j) = Y_{j,i} \qquad (2.3)$$

[2]

The functions $\tilde{W}_i$ and X_i denote any write operation, respectively any read or write operation (X_i) for cell C_i.

Hayes considered M_r to be an incompletely specified n-state Mealy automaton with $3r$ input symbols: $READ_i$, $WRITE_i$ 0 and $WRITE_i$ 1 for $i \in \{0..r-1\}$, and 0/1 as output symbols. The function z (2.4) describes the output function

[2] The orginal paper states $R_i(Y_j) = Y_j$, but this is probably a technical error.

of the automaton.

$$z(X_i, Y_j) = \begin{cases} y_{j,i} & \text{if } X_i = R_i \\ \text{---} & \text{if } X_i = \tilde{W}_i \end{cases} \tag{2.4}$$

Functions (2.1) to (2.3) describe the state transitions. Figure 2.2 shows a corresponding state diagram with all valid transitions for a two bit memory M_2.

A sequence S of n functions $f_{0 \cdots n-1} \in X$ is called an *input sequence* for M_r. When S is applied to the state Y_j of M_r, it leaves M_r in a final state Y_k. The functions $F_0 = \left\{W_i, \bar{W_i}, R_i, z\right\}$ with $i \in \{0 \ldots r-1\}$ completely describe the behaviour of M_r.

Let S be an input sequence. $z^*(S, Y_j)$ is the output of the Mealy automaton, when S is applied to M_r in state Y_j. The output is pruned of "—"s, so it contains only the output of $READ$s, i.e. $z^*(W_0 R_0 \bar{W_1} R_1 R_0) = 101$.

2.2.2 Pattern-Sensitive Fault Model (*Hayes*)

A *fault* is said to occur, when the functions $F_0 = \left\{W_i, \bar{W_i}, R_i, z\right\}$ change to $F = \left\{W_i^F, \bar{W_i^F}, R_i^F, z^F\right\}$ where $W_i^F, \bar{W_i^F}, R_i^F$ with $i = 0, 1, \cdots r-1$ are arbitrary mappings on $\{0,1\}^r$, and z^F is any mapping $\{0,1\}^r \rightarrow \{0,1\}$ [65].

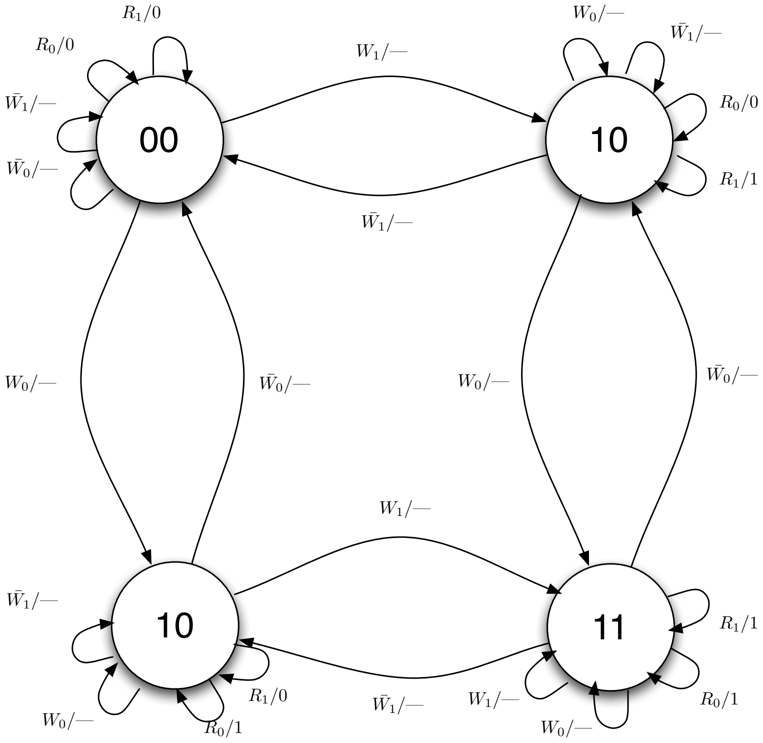
Hayes terms the fault-types this model describes as *Pattern Sensitive Faults (PSF)*.

The fault types stuck-at fault, coupling fault, and address decoding fault, are described in section 2.2.3.

The fault F changes the machine M_r, described by F_0 into M_r^F, described by F.

Hayes distinguished three different aspects of "state" of M_r^F: the *internal state* of M_r^F, the *expected state* of M_r^F, and the *apparent state* of M_r^F.

internal state The actual pattern of 0's and 1's stored in M_r^F, denoted as Y_j is called the internal state of M_r^F.



		Input					
		W_0	W_1	$\bar{W}_0$	$\bar{W}_1$	R_0	R_1
	00	10,—	01,—	00,—	00,—	00,0	00,0
State y_0y_1	01	11,—	01,—	01,—	00,—	01,0	01,1
	10	10,—	11,—	00,—	00,—	10,1	10,0
	11	11,—	11,—	01,—	10,—	11,1	11,1

Figure 2.2: *RAM state diagram for a memory with 2 cells M_2 [65].*
The four states $00 \ldots 11$ are all possible states of the mealy-automaton for a two bit RAM.
A state transition W_i denotes a $WRITE$ 1 operation into cell i, $\bar{W}_i$ is an analogous $WRITE$ 0 operation into cell C_i, and R_i reads the content for C_i.
The left cell is indexed by $i = 0$, the right cell by an index of $i = 1$.
For M_r the state table has 2^r rows and $|\{R, W, \bar{W}\}| * r$ columns, resulting in a fully connected Eulerian graph with 2^r states and $3 * r * 2^r = 3 * 2^{r+1}$ arcs.

expected state The expected state of M_r^F is Y_k, written as $E(Y_j)$. It is the state M_r^F would be in, if it were fault free.

apparent state $A(Y_j)$ is defined as $(z^F(R_0, Y_j), z^F(R_1, Y_j), \ldots, z^F(R_{r-1}, Y_j))$ and is termed the apparent state, the output of reading each cell $C_{0 \cdots r-1}$.

Let S be any input sequence that leaves M_r in the state Y_k, independently of the previous state of M_r^F. *Iff*, at some time t, the functions $E(Y_j)$ and $A(Y_j)$ differ for any internal state Y_j of M_r^F, the fault F is a *detectable* PSF.

2.2.2.1 PSF Fault Detection

Fault detection is conducted by feeding an input sequence S into M_r and M_r^F. If the output of feeding S into M_r and the output of feeding S into M_r^F differ, then a PSF has been detected. S is called a checking sequence, when it allows the detection of every fault described in the fault model.

Hayes proved, that the checking sequence needed to detect every PSF in M_r^F has a minimum length of $O\left(\left(3r^2 + 2r\right) 2^r\right)$.

2.2.2.2 Neighbourhoods

Current server and workstation configurations have memory configurations that include memory sizes in the area of several gigabits. A test algorithm with exponential runtime is clearly infeasible for this amount of memory. By restricting the fault model it is possible to significantly reduce the complexity of the test algorithm.

The first restriction Hayes introduced was the introduction of *local* PSFs (LPSF). In an unrestricted PSF, each $READ$ or $WRITE$ can affect, or be affected by every memory cell C_i. The reduction of test complexity is achieved by restricting the affected and affecting cells of an operation X_i to a *neighbourhood* N_i of C_i, with $C_i \in N_i$. Hayes justified this constraint to neighbourhoods by arguing that a PSF is likely to be caused by physical defects that affect adjacent cells or lines.

N, the set of all neighbourhoods, is defined as $N = \{N_0, N_1, \cdots, N_{r-1}\}$.

If N is a partition, that is, N consists of $k \leq r$ non-overlapping neighbourhoods N_i, it is called a *closed neighbourhood*, else *open neighbourhood*. A LPSF $F(N)$ that affects a closed neighbourhood is termed closed fault, respective open fault for an open neighbourhood.

Optimal fault detection in closed neighbourhoods To quote from the article:

> "Let $N = \{N_1, N_2, \cdots, N_k\}$ be a closed set of distinct neighbourhoods in M_r. Every closed fault $F(N)$ can be detected by a sequence $S = S_1 S_2 \cdots S_k$, where S_i is a checking sequence for N_i and $1 \leq i \leq k$. S is an optimal test for $F(N)$ if each S_i is an optimal checking sequence for N_i" [65].

By partitioning the r bit memory into α partitions of length $q = \frac{r}{\alpha}$ with $q, \alpha \in \mathbb{N}$, the test complexity of $O\left(\left(3r^2 + 2r\right) 2^r\right)$ for the detection of unrestricted PSF can be reduced to $O\left(\left(3q^2 + 2q\right) 2^q * \alpha\right)$.
Testing for PSF's in an open neighbourhood is more difficult but can be greatly simplified, when only one faulty memory element is allowed. Hayes called these faults Single PSF and provided an $O\left(\left(4q + 3\right) * 2^q * r\right)$-algorithm [65].

2.2.3 Fault Model by Nair *et al.*

The PSF fault model developed by Hayes is very potent, but ignores the experience that certain types of faults occur more often than others. This is aggravated by the fact that the algorithms to test for PSF-type faults require significant resources to test even modest memory sizes.
Nair *et al.* presented a simpler fault model, that draws from these insights and allows M_r to be tested in $O(r)$ [95].
Although the fault model presented by Nair *et al.* is functional, the described types of faults have their roots in the fact that certain hardware related faults occur more frequently than others. Because of this empirical background, and the possibility to efficiency test for these faults, Nair's fault model is still used – sometimes extended – today [102, 76].
In order to allow the definitions portrayed here to be easily recognized in Nair's work on the one hand, and to allow a comparison with the fault model of Hayes on the other hand, the next sections follow Nair's structure and nomenclature very closely and refer to related concepts in Hayes model where appropriate.
Nair's fault model is built on the idea of three functional blocks that form the memory. A separate fault model is defined for each of the functional blocks:

Memory Cell Array The memory cell array corresponds to the model of addressable binary cells discussed in section 2.2.1. The fault model Nair *et al.* proposed contains the following types of faults of the Memory Cell Array:

stuck-at fault One or more cells are stuck at 0 or 1. More formally this means that a given cell C_i is stuck at x when $z^*(W_i R_i, Y_j) = z^*(\bar{W}_i R_i, Y_j) = x$.

coupling fault One or more pairs of cells are coupled. Two cells (C_i, C_j) with $i \neq j$, are coupled, when a change in cell C_i from x to $\neg x$ causes a cell C_j to change as well. This relationship is not necessarily symmetric: a change in C_j might not affect C_i. Each cell can participate in more than one coupling, e.g. it is valid under the fault model that (C_i, C_j) form a coupled pair and that (C_i, C_k) form another coupled pair at the same time.

Decoder The decoder selects the memory cell for the requested address. Nair *et al.* treated decoder faults as faults in the memory cell array, if the fault does not change the decoder into a sequential circuit. They present two different faults that might affect the decoder:

wrong cell selection The decoder addresses a cell C_j instead of cell C_i, with $i \neq j$, or the decoder selects no cell at all. Nair *et al.* argued that, if no cell is selected, C_i behaves like suffering from a stuck-at fault.

multiple cell selection If an operation X_i affects other cells $C_j, \ldots$ besides C_i, it is said to affect multiple cells. According to Nair *et al.*, this is equivalent to a coupling-fault.

Reader/Writer logic According to Nair *et al.*, the Reader/Writer logic consists of the sense amplifiers, the write drivers and other supporting logic. They visualize stuck output lines of the sense amplifiers, or writer drivers as stuck-at faults. Shorts or capacitive coupling between data input/output lines are interpreted as coupling faults.

A memory M_r of r cells can be tested in $O(r)$, in, as they call it “about” $30 * r$ tests.

2.2.3.1 Extended Fault Model

Nair *et al.* extended their fault model with k-coupled faults. A k-coupling fault is a fault, where the change of one cell C_i in a set of k cells C^k causes a change in $C_j \in C^k$ when the other $k - 2$ cells in C^k have some fixed values. Comparable to Hayes’ neighbourhoods, Nair *et al.* restricted the k-sets to be non-overlapping. These restricted k-coupled faults are less complex to test, and they provide an $O(r * log_2 r)$ algorithm for restricted 3-coupling faults. No efficient algorithm for $k > 3$ is presented in the paper.

2.2.4 How Fault Models Relate to this Thesis

The two fault models presented in the previous sections are not the only fault models for RAM. Many more are published, e.g. [49, 48, 69, 84, 45, 118, 76, 111, 55], or held as trade secret by semiconductor manufacturers.

Different fault models and memory tests often aim for different goals: Hayes presented a very thorough fault model, that can only be tested with significant resources. Nair, on the other hand, presented a fault model that can be tested efficiently, but detects fewer errors. The test complexity and comprehensiveness of the published fault models is distributed between linear, quadratic and exponential complexity. Riedel and Rajski published a research paper that compares several fault detection algorithms and their respective fault models [102].

Choosing the *right* fault model is making a trade-off decision between performance and thoroughness. An ideal solution would be, that the fault model used by the memory test is exchangeable, so that different models can be selected, depending on the needs.

2.3 Fault management

Defective main memory is not the only source of data corruption. Different research groups analysed various components of computer systems including disk storage[110, 43], and CPU caches[93], and found that they may corrupt data as well. Even without data corruption, physical and software systems alike are subject to faults.

Large scale systems reach complexity levels that are so high, that errors that seldomly occur in modestly sized systems occur fairly frequent in these complex systems. An enlightening example for hardware complexity is one of the primary design goals for the BlueGene supercomputer[37]:

> The desired MTBF of the system is at least 10 days.

Besides hardware induced faults, software is seldom error free. A good example for the inherent complexity of software is the quicksort algorithm: Created by C. A. R. Hoare in 1961[68], incorrect implementations that handle corner cases wrong are still found.

Fault management is the process of responding to *detected* faults. The correct response to a fault heavily depends on the context the fault happens in, and the policy set in place for the occured class of errors.

A common approach to fault handling is the *fail fast*-policy: Upon the detection of an error condition, the system fails and stops to operate as quickly as possible.

2.3.1 Generic fault management architecture

Fail fast is not always the appropriate response to a fault. Some faults, such as a unavailable resource, can easily be handled by retrying the action, probably with an exponential backoff to reduce congestion caused by frequent retries.

Other faults require more or less complex counter actions, especially when they affect workflows in distributed systems. Workflow and business related fault management are treated by other projects[38], and are not subject of this thesis.

Fault management in the context of operating systems deals with faults in hardware, and entities directly managed by the operating system, mostly processes. A specific instance of a fault management architecture, "Predictive Self Healing" for Solaris 10, is described in section 3.2.1. Section 2.3 takes a more abstract view on the subject of fault management, and specifically relates fault management to memory errors.

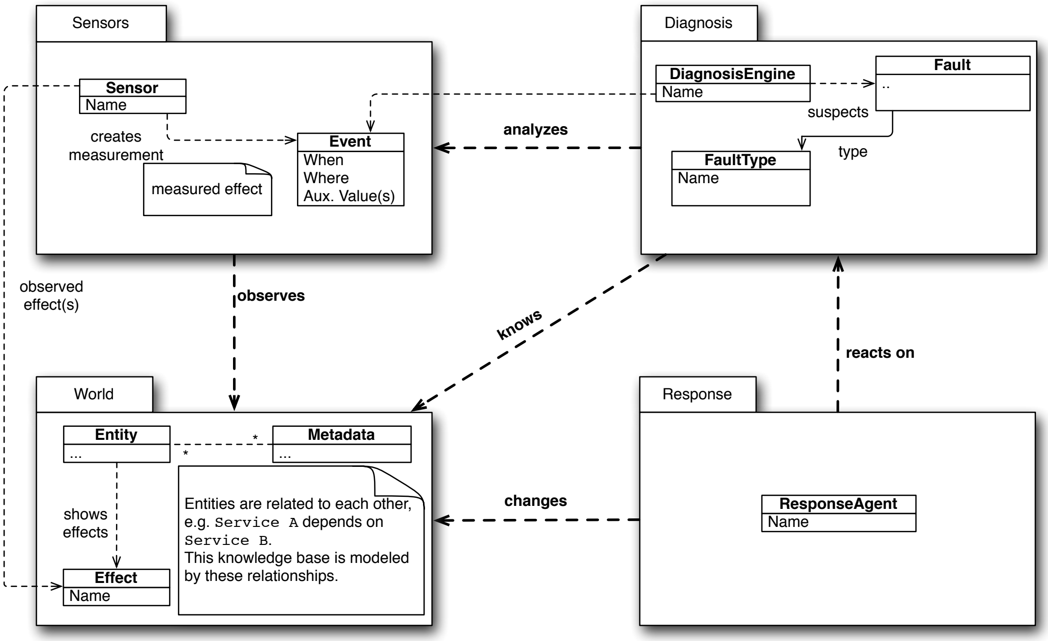


Figure 2.3: *Abstraction of a fault management architecture*
This figure depicts the class diagram of an idealised fault management architecture.
The box labeled *World* in the bottom left corner represents the managed system, and every FMA related aspect of it, including human operators, etc. Entities show measurable effects like hardware temperature, or CPU usage. Everything that is observable is understood as an effect.
Central to an FMA is a knowledge base of the relationships between the hard- and software components forming the managed system.
Sensors, on the top left, measure the effects produced by the world.
Diagnosis engines interpret the measurements of the systems and diagnose faults.
Faults are then handled by *response* agents that act upon the system.

An abstraction of an FMA convenient for further discussion is shown in figure

2.3. The presented abstraction aims to separate the aspects of data collection, fault detection by analysing the collected data, and fault handling into distinct aspects. Four distinct subsystems represent these aspects and form this architecture:

World The world is everything that either directly affects the operating system, e.g. hardware, external influences or artifacts that are affected or managed by the operating system, e.g. processes, filesystems, devices. These *entities* show *effects* like, but not limited to, logfile entries, disk activity, process creation and termination, signals and messages, et cetera.

Sensors Effects arising in *the world* can be measured by *sensors*. A *sensor* is primarily defined by its software component. A good example for a sensor is a temperature sensor. It consists of a hardware device that measures the temperature (effect) of something else (an entity, e.g the processor) and a software component that provides these measurements (events) to clients.

Diagnosis Staying with the example of a temperature sensor, a diagnosis unit interprets the measurements provided by the sensors. If the temperature is too high, the diagnosis unit suspects a *fault*. The diagnosis unit is not limited to one sensor or one type of fault. More complex reasoning can be done by interpreting chassis temperature, processor load, the voltage of the processor fan and finally the temperature of the processor. Correlating these factors could lead to several faults being diagnosed, e.g. "processor temperature too high" and "processor fan out of order".

Diagnosing faults is a process that utilizes different fault models for the computer system monitored to detect or predict faults.

Response Ideally, detected faults trigger responses. A reasonable first response to the faults detected in the last step of the response agent would be to lower the processors frequency, or to schedule tasks running on that processor to a different processor to take the load of the processor. A failed fan probably needs quick human intervention, so paging an operator is another response to the fault.

Responses may influence the *world* and trigger new effects, causing new events to be recorded, and maybe new faults to be diagnosed.

2.3.1.1 Example

The following example shows how two related subsystems work together to compensate a hardware fault. The first system is the *Service Management*, it knows the dependencies between different services and knows, how to start and stop the installed services. The second system is the *Fault Management Architecture*.
Suppose the Service Management knows, that $service_A$ depends on $service_B$ and needs to be restarted every time $service_B$ is restarted.
The following chain of events show how the unexpected termination of $service_B$ leads to a degraded system state that is automatically repaired by the fault-, and the service management. It also shows that a sophisticated fault management implementation needs to know a substantial amount of meta information covering dependencies between hardware and software components, and potential recovery measurements.

1. The fact that $service_A$ depends on $service_B$ and that $service_A$ needs to be restarted when $service_B$ is restarted is made known to the service management.

2. The *user* instructs the system to start $service_A$.

3. The *service management* recognizes that $service_B$ needs to be started and initiates the startup of $service_B$.
   $service_A$ is left in a $waiting\_for\_dependencies$ state.

4. When $service_B$ is up, the *service management* continues the startup process of $service_A$.

5. $service_A$ is now in a *clean* state and ready to serve its clients.

6. The memory controller reports an uncorrectable error, causing an exception that triggers the *memory failure sensor* to record this effect.

7. The *diagnosis* subsystem detects that $service_B$ is affected by the error.
8. A *response agent* kills $service_B$ and offlines the affected memory frame.
9. $service_B$ terminates abnormally.
10. The *fault management* detects that $service_B$ terminated.
11. According to the configuration made in the first step a *diagnosis expert* recognizes a fault for $service_A$.
12. A *response agent* instructs the *service management* to shut down $service_A$ and to initiate process described in step 3.
13. The fault has been recovered.

2.3.2 Fault Trees

Describing and analysing the impact that certain faults have on a system can be a daunting task. Fault Trees describe how various events within the system may lead to undesired events [63], and greatly help to estimate the impact faults have on a system. First introduced in 1961, fault tree analysis is nowadays an important tool in risk management.

Figure 2.4 shows a basic fault tree for undetected memory errors. Depending on the importance, and the complexity of the problem analysed, fault trees can get very large and very complex. Haasl presented two approaches to fault tree analysis that he called *Primary-* and *Secondary Failure Technique*. A primary component failure is a failure, that happens, when the component showing the failure is operating within its specification. A secondary component failure is a failure that happens while the component is subject to enviroental stress, such as can be caused by the failure of other components in the system [63]. If only primary component failures are included in the fault tree, the *Primary Failure Technique* is used. For more critical systems, the *Secondary Failure Technique* can be used. It extends the former technique by including secondary component failures, and is substantially more complex.

By assigning probabilities to faults in leaf nodes, the probability of top level faults can be calculated, compare to [82], [96], and many more publications concerning fault tree analysis. For a discussion on fault tree analysis of computer systems see [101].

Fault trees are an important tool for a FMA. An example for the usage of fault trees is presented in section 3.2.1.

2.3.3 Error Reporting of Memory Errors

It would be beneficial if the ECC hardware would report detected memory errors to the operating system. The Intel *x86* architecture supports a mechanism termed Machine Check Exception (MCE) that allows the processor to report different hardware error conditions to the operating system [9, Chapter 15]. The format and content of the different exceptions is highly platform specific and needs specialised decoders to normalise them.

Recent Intel processors (starting with 45nm Intel 64 processors a with CPUID signature *DisplayFamily_DisplayModel* encoding of $06H\_2EH$) allow software to "perform recovery action on a certain class of uncorrected errors and continue execution"[9, Chapter 15.6]. The Linux kernel supports this recovery for ECC-errors with the HW-Poison patches[60].

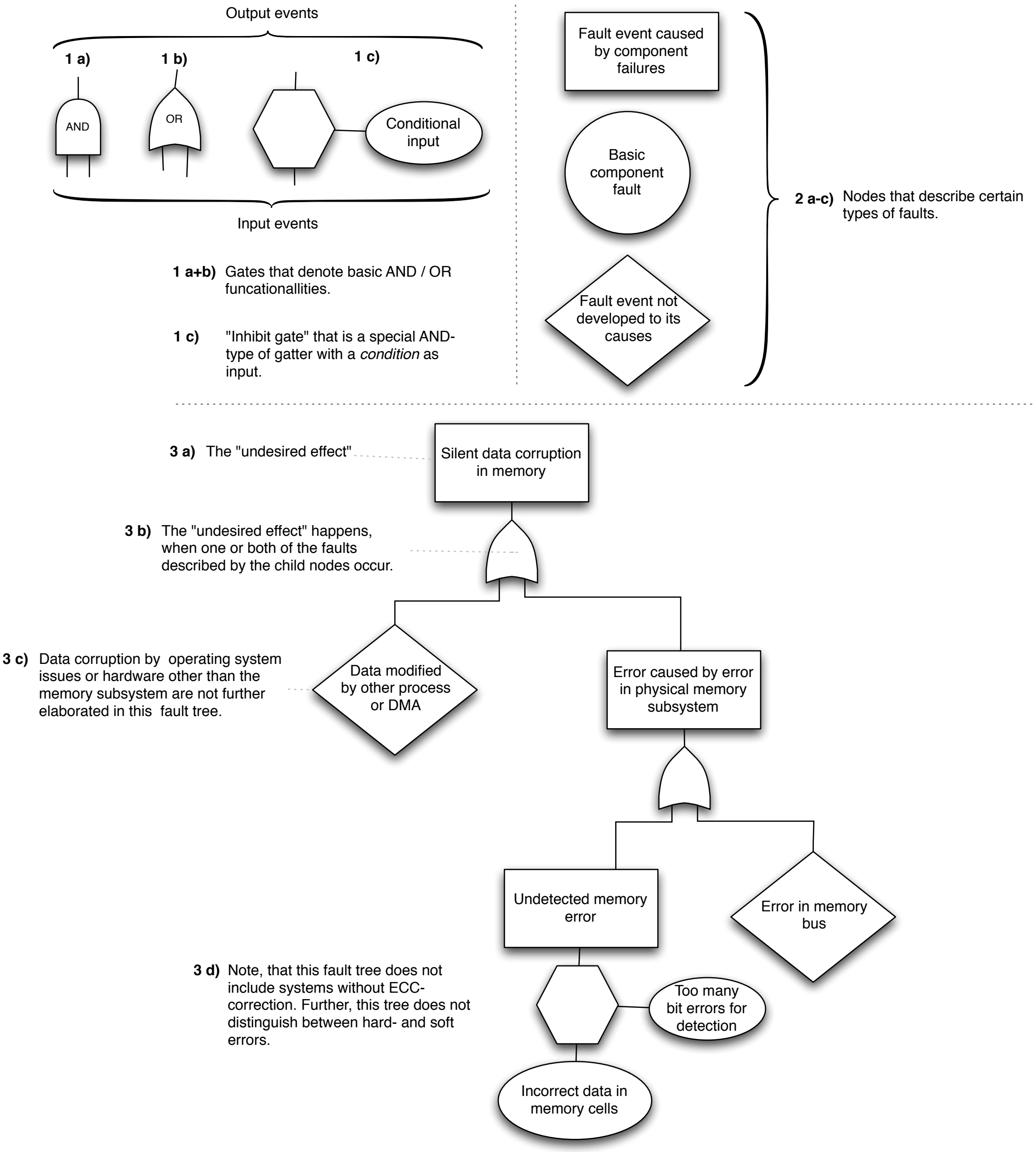
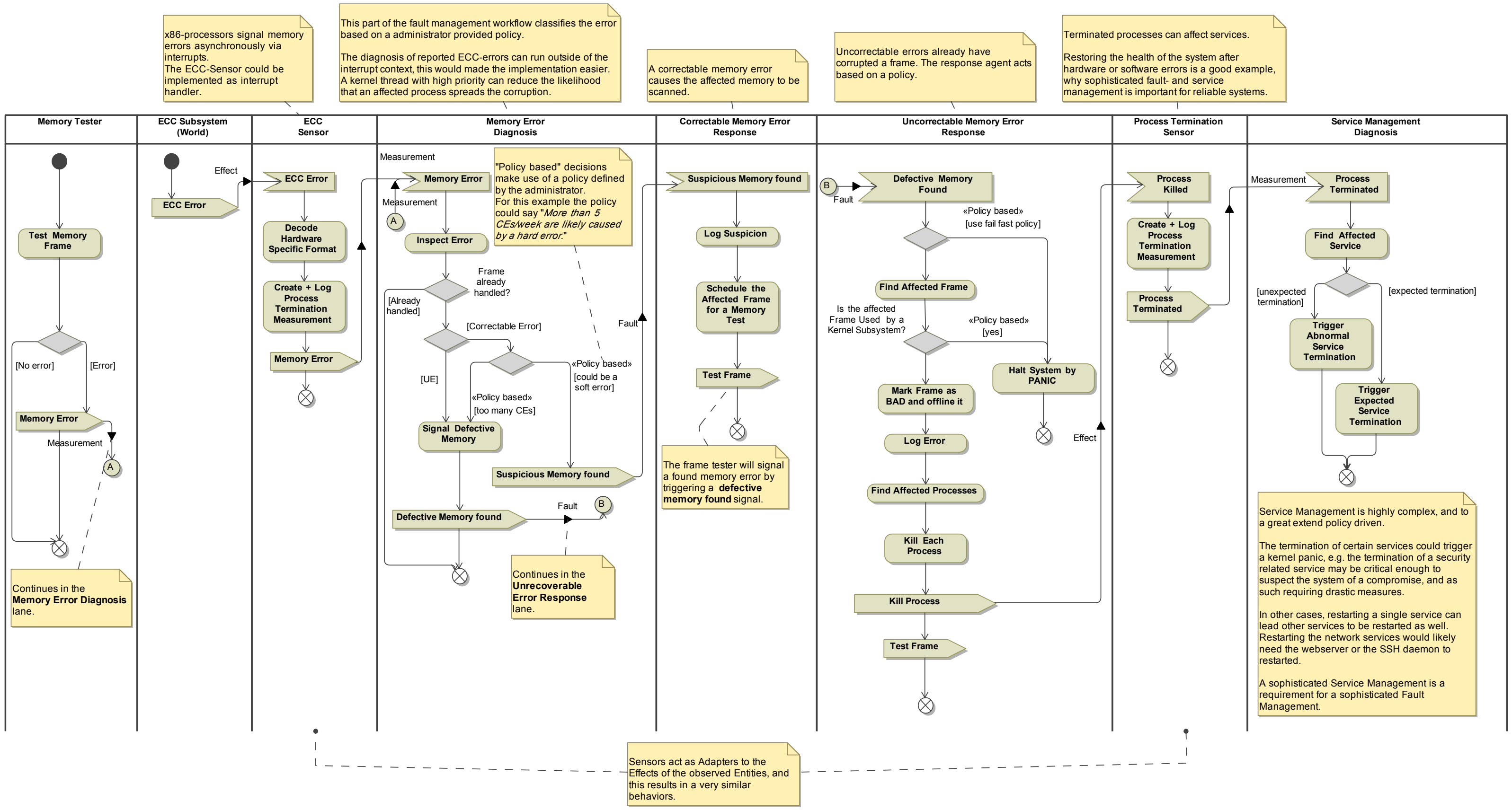

Output events
1 a)
1 b)
1 c)
AND
OR
Conditional input
Input events
1 a+b) Gates that denote basic AND / OR funcationallities.
1 c) "Inhibit gate" that is a special AND-type of gatter with a *condition* as input.
Fault event caused by component failures
Basic component fault
Fault event not developed to its causes
2 a-c) Nodes that describe certain types of faults.
3 a) The "undesired effect"
Silent data corruption in memory
3 b) The "undesired effect" happens, when one or both of the faults described by the child nodes occur.
3 c) Data corruption by operating system issues or hardware other than the memory subsystem are not further elaborated in this fault tree.
Data modified by other process or DMA
Error caused by error in physical memory subsystem
Undetected memory error
Error in memory bus
3 d) Note, that this fault tree does not include systems without ECC-correction. Further, this tree does not distinguish between hard- and soft errors.
Too many bit errors for detection
Incorrect data in memory cells

Figure 2.4: *Basic Fault Trees*
This figure shows basic fault trees as they are described by Haasl [63].
A basic fault tree has an "undesired event" as root. Child nodes denote events or conditions that trigger the parent node. In the most basic form of fault trees, child nodes are either gates (AND, OR (**1a+b**)) that combine multiple inputs into one output, or different types of fault events (**2a-c**)).
A simplified fault tree for *undetected memory errors* is shown in **3 a-d**)).



x86-processors signal memory errors asynchronously via interrupts.
The ECC-Sensor could be implemented as interrupt handler.

This part of the fault management workflow classifies the error based on a administrator provided policy.

The diagnosis of reported ECC-errors can run outside of the interrupt context, this would made the implementation easier.
A kernel thread with high priority can reduce the likelihood that an affected process spreads the corruption.

A correctable memory error causes the affected memory to be scanned.

Uncorrectable errors already have corrupted a frame. The response agent acts based on a policy.

Terminated processes can affect services.

Restoring the health of the system after hardware or software errors is a good example, why sophisticated fault- and service management is important for reliable systems.

"Policy based" decisions make use of a policy defined by the administrator.
For this example the policy could say "*More than 5 CEs/week are likely caused by a hard error.*"

Continues in the **Memory Error Diagnosis** lane.

Continues in the **Unrecoverable Error Response** lane.

The frame tester will signal a found memory error by triggering a **defective memory found** signal.

Service Management is highly complex, and to a great extend policy driven.

The termination of certain services could trigger a kernel panic, e.g. the termination of a security related service may be critical enough to suspect the system of a compromise, and as such requiring drastic measures.

In other cases, restarting a single service can lead other services to be restarted as well. Restarting the network services would likely need the webserver or the SSH daemon to restarted.

A sophisticated Service Management is a requirement for a sophisticated Fault Management.

Sensors act as Adapters to the Effects of the observed Entities, and this results in a very similar behaviors.

Figure 2.5: *Exemplary handling of a detected ECC-fault*
This UML Activity Diagram sketches, how the architecture from figure 2.3 can be utilized to handle memory errors.Two different events can trigger the process: A memory error found by a software based memory tester, or a memory error detected by the ECC-hardware. On systems with ECC-hardware a race condition between memory test and ECC-hardware is possible, when – as is the case for x86 systems – ECC errors are reported asynchronously. The **Memory Error Diagnosis** take this into account, by testing for frames that are already handled by the fault management.
This diagram shows, how Fault- and Service management are interconnected: The forced termination of a process triggers the **Service Management Diagnosis** to recover the service.

2.3.4 Handling Memory Errors

Once a memory error is reported, the FMA is responsible for handling the error. The abstractions introduced in section 2.3.1 are used to discuss how CEs and UEs could be handled by a fault management implementation. Figure 2.5 shows how memory error detection, diagnosis, and handling influence each other.

Several things need to be done, when a memory error has been detected by the ECC hardware. In case of an UE, processes that use the frame have to be informed, and probably killed to prevent the corruption caused by the error to spread further. The same applies to kernel subsytems that use the frame. Handling a CE is easier, because no corruption has happened yet.

After these immediate actions have been performed, decisions about the actual state of the frame must be made. The error could have happened by random chance, for example by cosmic radiation. It is also possible, and the field study realized by Google suggests that it is more likely, that a physical defect of the memory module caused the error.

The process of handling an immediate error and the process of deciding what to do in the long term should be distinguished. Both processes have different contexts and different objectives. The objective for the short term handler is to prevent that corruption caused by the memory error is spread. The runtime context is probably a machine check handler running in an interrupt context, that reacts upon a machine check exception from the memory controller. Thus the short time handler has to act fast and without any human intervention. In figure 2.5, the short term handling is implemented by several components in concert. **ECC Sensor**, **Memory Error Diagnosis**, and the **Unrecoverable Memory Error Response** are steps that need to run as quickly and as seamlessly as possible to prevent corruption from spreading.

After the direct danger of spreading corruption has been removed, the medium term handler needs to decide what to do next. In contrast to the short time handler the medium term handler has more time and more resources at its disposal. It runs decoupled from the short time handler and is probably implemented as a user process. The objectives of the medium term handler are located at a higher level and include sophisticated error detection and reporting. In figure 2.5, the swim-lane labeled **Correctable Memory Error Response** is an example for a medium term handler. No immediate action is required, when a CE has been detected. Still, it is prudent to verify the neighbourhood surrounding the memory location that showed the CE.

2.3.4.1 Policies

What the correct treatment for a memory error is, depends on the context and type the error. In chapter 1 it has been motivated, that different usage scenarios for computer systems require different levels of confidence, and performance. The processes in figure 2.5 contain several decisions that can be configured by policies. This allows the behaviour of fault management to be configured without explicitly changing the software itself.

2.3.5 How Fault Management Relates to this Thesis

One of the objectives of this diploma thesis is to discuss, how Linux can be extended with a FMA. Section 2.3 described an idealised FMA and laid out, how this FMA can react to faults in memory.

Several candidates for components of this architecture exist. Candidates for the *Service Management* are the Upstart-project [35] originally developed for Ubunbtu or the SMF developed for Solaris 10 [104]. Detected memory faults can be handled by the HW-poison patches [60], that allow defective frames to be offlined and the mcelog program, that listens for memory faults reported by the hardware [30].

2.4 Using the Linux Kernel

In the first chapter it is stated, that the feasibility of a software based memory tester will be analysed by implementing such a memory tester for the Linux-kernel. This section is split into three parts: A justification for the usage of Linux as implementation target, an introduction into the Linux mm subsystem, followed by a description of the current state of fault management in the Linux-kernel and distributions.

The Linux-kernel itself has been ported to many different architectures, from embedded systems to clustered high performance systems [23, 12, 87, 89]. The development process itself has been driven by a multitude of interest parties as well, the list of contributors includes companies and organisations like Sun, Intel, SuSE, RedHat, the US American National Security Agency (NSA), and many more: Between 2005 and 2008 “over 3700 individual developers from over 200 different companies have contributed to the kernel” [27]. This lead to the highly adaptive and complex system Linux is now. Section 2.4.3 presents statistics and statements from the kernel documentation that illustrate how volatile the kernel code in general and the mm-code specifically really is.

2.4.1 Why Use the Linux Kernel?

The operating system that is used as implementation target for the thesis' online memory tester has to fulfill a number of requirements. First of all, it must be possible to access most of the physical memory installed. This requirement is probably fulfilled by all mainstream operating systems for low- to midrange servers. Linux on x86 can manage nearly all memory the CPU can access [12].

Secondly, and for several reasons, the operating system selected should be in wide use. Selecting an operating system that is not widely used limits the body of knowledge available, as there are, for example, fewer books and fewer experts. Although this is not strictly an academic argument, the number of possible users for the memory tester is lower, when a rarely used operating system is selected. A higher number of users may improve the chances, that further academic research will be based on this work, such raising the significance of this diploma thesis.

For practical reasons, the operating system selected should run on readily available hardware. Further, it should be open source, so that the operating system itself is malleable and can be modified, when necessary. Publishing fragments of open sourced software reduces the risks that the publication of excerpts of proprietary, closed source code poses. Lastly, the author's existing experience, and knowledge influences the choice as well.

2.4.1.1 Choosing Linux

The Linux-kernel is at the core of many different distributions, for example *Gentoo, Ubuntu, SuSe*, and *RedHat* [5, 8, 7, 6], and is published under the GNU Public License (GPL). As such it is open source and can be modified and extended to any extent. A number of books have been published on the subject of Linux kernel development in general [46, 53, 87, 89, 103], and there exists a book specifically treating Linux memory management – even if it was written before the current 2.6 series of the Linux kernel was released [62].
The Linux homepage `http://www.kernel.org` describes Linux as follows:

> "Linux is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX and Single UNIX Specification compliance. It has all the features you would expect in a modern fully-fledged Unix, including true multitasking, virtual memory, [. . .], proper memory management, [. . .]. Although originally developed first for 32-bit x86-based PCs (386 or higher), today Linux also runs on (at least) [. . .] AMD x86-64, [*and many more*] architectures; for many of these architectures in both 32- and 64-bit variants. Linux is easily portable to most general-purpose 32- or 64-bit architectures as long as they have a paged memory management unit (PMMU) [. . .]" [12]

Together with the large number of academic Linux users – in may 2010, searching for the keyword "Linux" in the ACM digital library yielded 17,227 results – and with the strong presence of Linux in the targeted low- to mid-range server ecosystem, Linux complies to all requirements stated.
For similar reasons, the Intel x86 family of processors is used as underlying hardware: This architecture is predominant in the low- to midrange server market, and readily available to developers.

Certain important features like MCE were only available in the 64-bit version of the Linux kernel, although they have been back ported to recent 32 bit kernels. Restricting the implementation to the 64-bit kernel does not greatly reduce the meaningfulness of the implementation. Many server systems run a 64-bit kernel, so evaluating the runtime and coding impact of the implementation on a 64-bit kernel is more significant than for the 32-bit kernel.

2.4.2 The Linux Memory Management

The Memory Management subsystem of the Linux kernel is often called the *VM* for Virtual Memory. This term expresses, what the Linux memory management does at its core: It virtualizes physical memory, thereby abstracting from the underlying hardware.

The virtualization is not only restricted to the basic hardware aspects like the total amount of addressable memory, the memory topology provided by the hardware, or processor architecture specifics. The mm also manages the virtual address space for of the kernel and user space processes, and many things more.

The details described in this section have been extracted from the kernel source, the kernel's homepage [12], and various books and papers including [46, 53, 62, 87, 89, 103, 42, 23, 60].

The following sections describe the tasks and structure of the Linux mm to the extend, that all important aspects that relate to a memory test are covered. Figure 2.6 on page 37 shows an overview, of how several components of the mm and the hardware relate to each other.

2.4.2.1 Tasks of the Linux MM

Virtually every service the kernel provides uses the memory subsystem. This central role requires the memory subsystem to work as efficient, and as reliable as possible. Besides the demand to provide these services as efficient as possible, the memory subsystem has to work on different processor and memory architectures that provide different mechanisms to handle physical memory. This calls for an implementation design that can abstract from the actual hardware. The amount of memory handled ranges from systems, like embedded routers, with

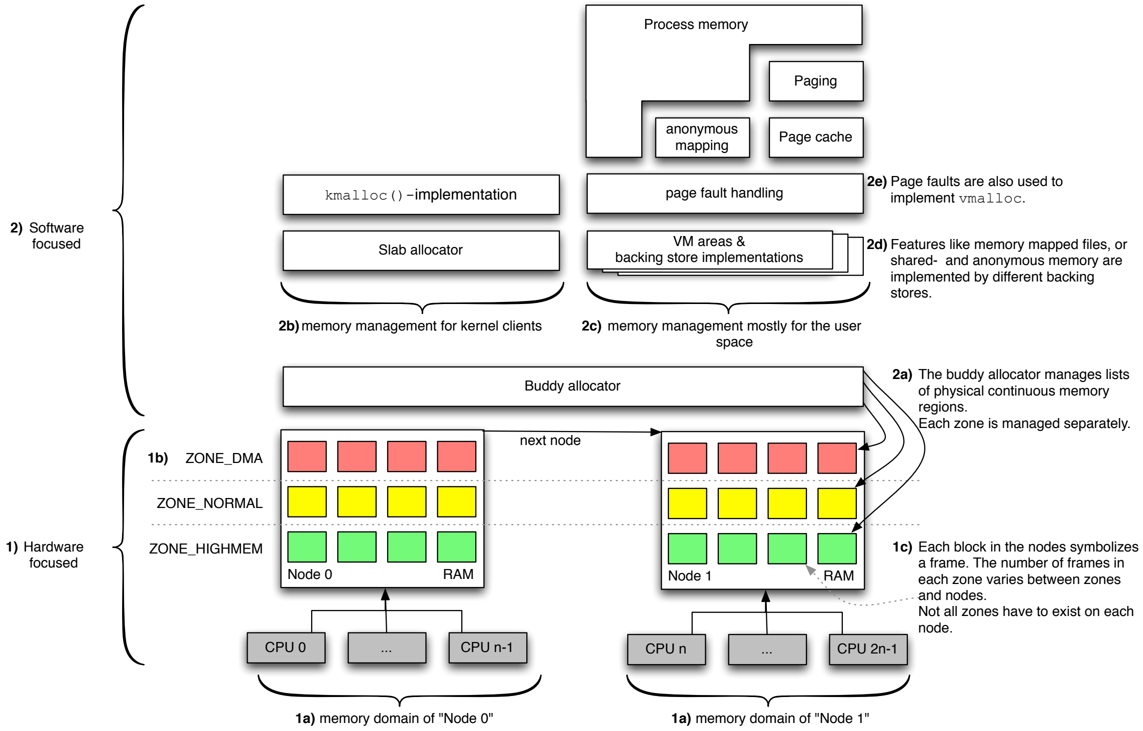

Process memory
Paging
anonymous mapping
Page cache
kmalloc()-implementation
page fault handling
2e) Page faults are also used to implement vmalloc.
2) Software focused
Slab allocator
VM areas & backing store implementations
2d) Features like memory mapped files, or shared- and anonymous memory are implemented by different backing stores.
2b) memory management for kernel clients
2c) memory management mostly for the user space
Buddy allocator
2a) The buddy allocator manages lists of physical continuous memory regions.
Each zone is managed separately.
1b) ZONE_DMA
next node
ZONE_NORMAL
1) Hardware focused
ZONE_HIGHMEM
Node 0
RAM
Node 1
RAM
1c) Each block in the nodes symbolizes a frame. The number of frames in each zone varies between zones and nodes.
Not all zones have to exist on each node.
CPU 0
...
CPU n-1
CPU n
...
CPU 2n-1
1a) memory domain of "Node 0"
1a) memory domain of "Node 1"

Figure 2.6: *Overview of the Linux mm.*
The Linux mm is a layered architecture that consists of different components. The usage-of or depends-on relationship is top-down, e.g. the process memory depends on the handling of page faults.
The hardware focused concepts are shown the bottom of the image (**1a-c)**). The system in this example features $2n$ processors on two NUMA nodes (**1a)**). Physical memory is partitioned into memory zones (**1b)**), and each zone is partitioned into frames (**1c)**).
The software components in **2)** abstract most hardware related aspects. **2a)** The most basic memory management is implemented by the buddy allocator, which manages areas of consecutive frames. Clients request memory in sizes of 2^{order} frames, where $0 \leq order \leq MAX\_ORDER$.
2b) The slab allocator manages objects of a fixed size, e.g. file descriptors, in caches that are themselves allocated from the buddy allocator. The kernel *kmalloc* implementation then is based on the slab allocator.
2c) Process memory and disk caching is based on a number of abstraction mechanisms and mm-subsystems. Most of these services are based on the concept of address spaces. address spaces are a central abstraction of the mm and are implemented by different subsystems called backing stores.

a very constrained amount of memory to systems, like supercomputers, that have to handle several TiB of RAM [23, 62, 12].

Together this makes the mm one of the most complex, and most performance tuned parts of the Linux kernel [89]. Section 2.4.3 elaborates on the complexity, and the volatility of the mm-subsystem implementation.

The tasks of the Linux memory management can be broken down into four main services:

1. Detect and manage the available physical memory frames.
2. Efficiently manage the allocation of large contiguous and of non-contiguous memory areas.
3. Efficiently manage the allocation of small memory areas.
4. Efficiently and securely manage process memory.

Figure 2.6 summarizes, how these services relate to each other.

The detection of available memory is a very hardware dependant task and will not be discussed here, because it is not important for a memory test that builds on top of an operating system. What is important though, is the relation of memory frames to physical memory modules. This is discussed in section 2.4.2.10. Further hardware related topics are discussed in section 2.4.2.3, that discusses how the Linux kernel manages Uniform Memory Access (UMA) and NUMA architectures.

The next section discusses virtual memory, a very fundamental and powerful abstraction mechanism used by the Linux mm.

2.4.2.2 Virtual Memory

At the core of the Linux mm lies the virtualization of memory. Instructions that execute on the processor do not directly access physical memory. The addresses used by instructions are virtual, not physical, addresses. Before the actual RAM is accessed by an instruction, the virtual address is translated into a physical address. This translation step is done at runtime, and with the help of the processor's Memory Management Unit (MMU)[3]. The decision, which virtual address is mapped to which physical memory location is made by the Linux kernel. Mapping each virtual address separately to a physical address would be very inefficient. On processors like the Intel x86 family, Linux uses a concept called frames. The available physical memory is split up into equal sized frames $Frame_{0..max\_pfn-1}$ of regions of contiguous physical memory. Each region spans a range of 2^{PAGE_SHIFT} addresses, and the memory is byte-addressable. The physical memory regions are called page frames, or just frames.

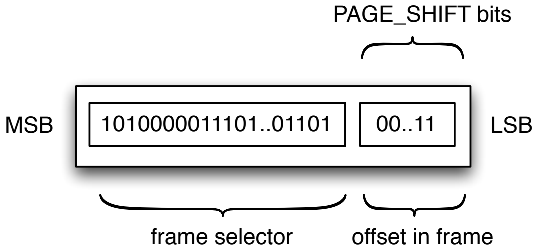


Figure 2.7: *Virtual Address.*
This image shows, how the virtual address a_v can be visualised as consisting of a selector for the memory frame and an offset within the selected frame.

Thus, a virtual address a_v can be visualised as consisting of a selector for a frame, and an offset within the frame. Figure 2.7 shows, how a virtual address is split up.

A (virtual) address space spans a range of $0 \cdots 2^n - 1$ virtual addresses, where n is 64 for the 64 bit kernel on the x86-64 architecture. Address spaces are used at the core of the Linux mm to manage memory for processes and the kernel itself.

Figure 2.8 on page 41 depicts, how address spaces and physical memory relate to each other. The address spaces of two processes are shown at the top and at the bottom of the image, the physical frames are located in the middle, arrows denote the mapping of a page in the address space onto a physical memory

[3] With certain restrictions, Linux can run on devices without MMU [10]. These architectures are used in embedded systems and not in server systems. For this reason they are not covered by this diploma thesis.

frame.

Virtualizing memory has several benefits, and some of these benefits are shown in the figure. By decoupling the virtual memory from the physical memory, frames can be shared between address spaces. In this example, the kernel code and text (shown in blue) is mapped into both processes. Another frame is shared between two processes, either as a means for inter-process communication, or to reduce the number of frames with duplicate content. The later is used by the operating system to transparently share the TEXT segments of libraries between processes. The implementation of UNIX process forks in Linux prevents the copying of the whole address space of the forked process by implementing a copy-on-write scheme [46]. Recent developments for the Kernel Virtual Machine (KVM) also periodically scan the RAM for duplicated frames [42, 59].

Figure 2.8, points **4b)** + **4c)** show two pages that are not backed by physical memory. In most circumstances, nearly all pages from a processes address space will *not* be connected to a physical frame. This stands to reason, because a system with an n-bit addressing scheme can normally only address and manage 2^n bytes of memory, but Linux provides each process with a virtual address space that has a size of up to 2^n bytes [62, 87, 89].

The figure hints at how the mapping between pages and frames is established: When a process accesses a virtual address that belongs to a page that is not mapped to a physical frame, a page fault is raised. Linux handles this page fault differently, depending on what exactly has caused the page fault.

2.4.2.3 UMA and NUMA

The memory access model for small Symmetric Multi-Processing (SMP) systems is often designed as an UMA-architecture. Each processor core can access every cell in the RAM at the same cost. Figure 2.9 shows an UMA system that is reduced to the components of CPU, RAM, and the memory bus that connects CPU and RAM.

Architectures based on UMA can be detrimental, if many processors access the memory. Bottlenecks can lie in the memory bus that connects the processors and the memory modules or in the memory modules themselves. E.g., the widely used Double Data Rate Synchronous Dynamic Random Access Memory

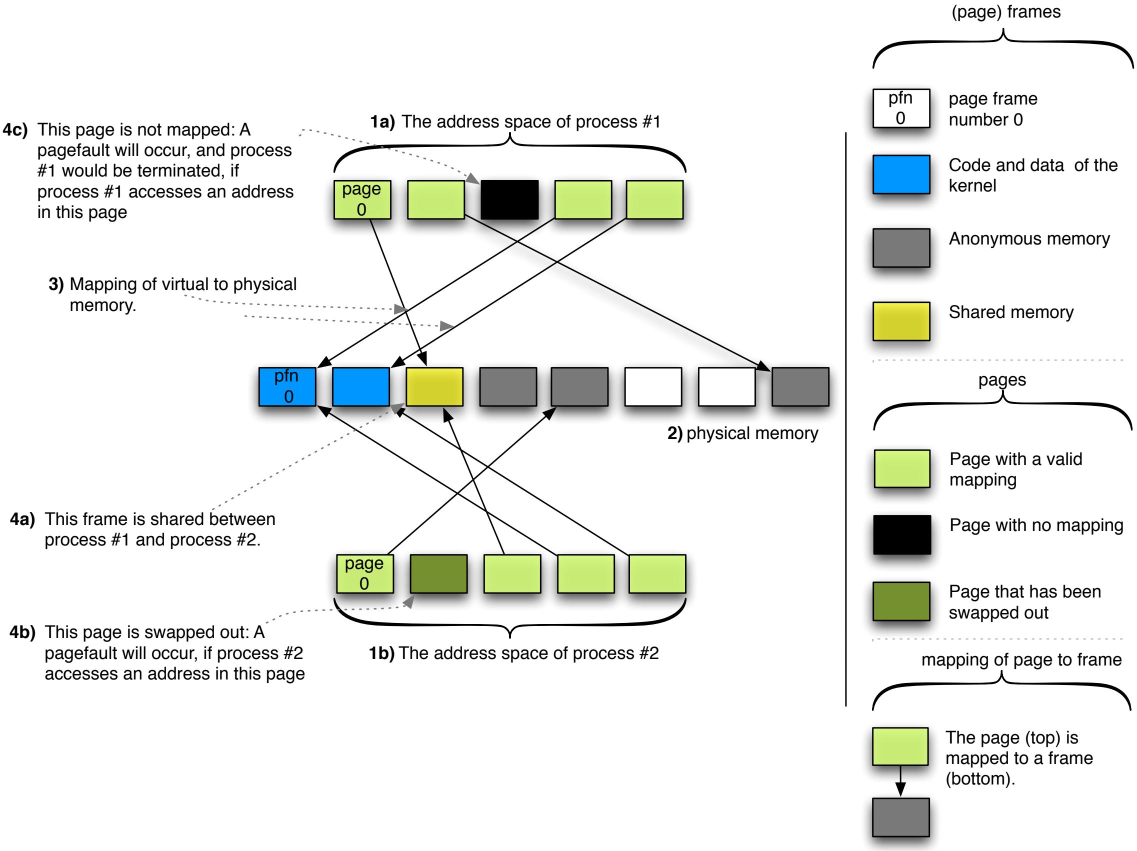

(page) frames
pfn 0
page frame number 0
Code and data of the kernel
Anonymous memory
Shared memory
pages
Page with a valid mapping
Page with no mapping
Page that has been swapped out
mapping of page to frame
The page (top) is mapped to a frame (bottom).
4c) This page is not mapped: A pagefault will occur, and process #1 would be terminated, if process #1 accesses an address in this page
1a) The address space of process #1
page 0
3) Mapping of virtual to physical memory.
pfn 0
2) physical memory
4a) This frame is shared between process #1 and process #2.
page 0
4b) This page is swapped out: A pagefault will occur, if process #2 accesses an address in this page
1b) The address space of process #2

Figure 2.8: *Virtual Address Spaces.*
This image shows, how address spaces are mapped to physical memory (**2)**).
Each of the two processes has its own virtual address space (**1a, b)**), that is mapped (**3)**) onto physical memory.
A few things are noteworthy:
The kernel resides in the first few page frames and each process maps the kernel at the end of its address space. Both processes share one frame, but process #1 has it mapped to a different page than process #2 (**4a)**).
A page of process #2 has been swapped out (**4b)**). If process #2 accesses an address in this page, a fault will be generated, and the page would transparently to the process be connected to a frame with the then swapped-in data.
4c) shows a page that has no mapping, accessing this page would cause the kernel to kill the process with a segfault.

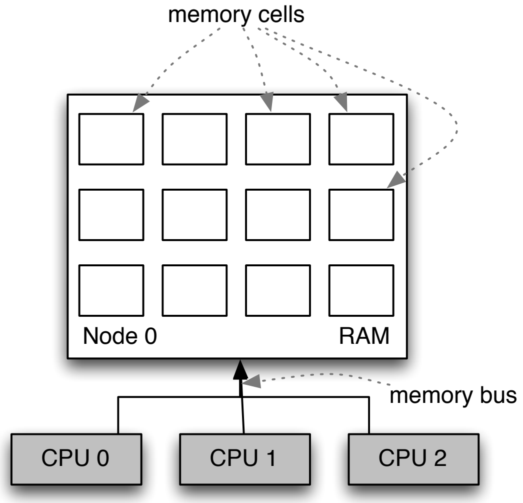


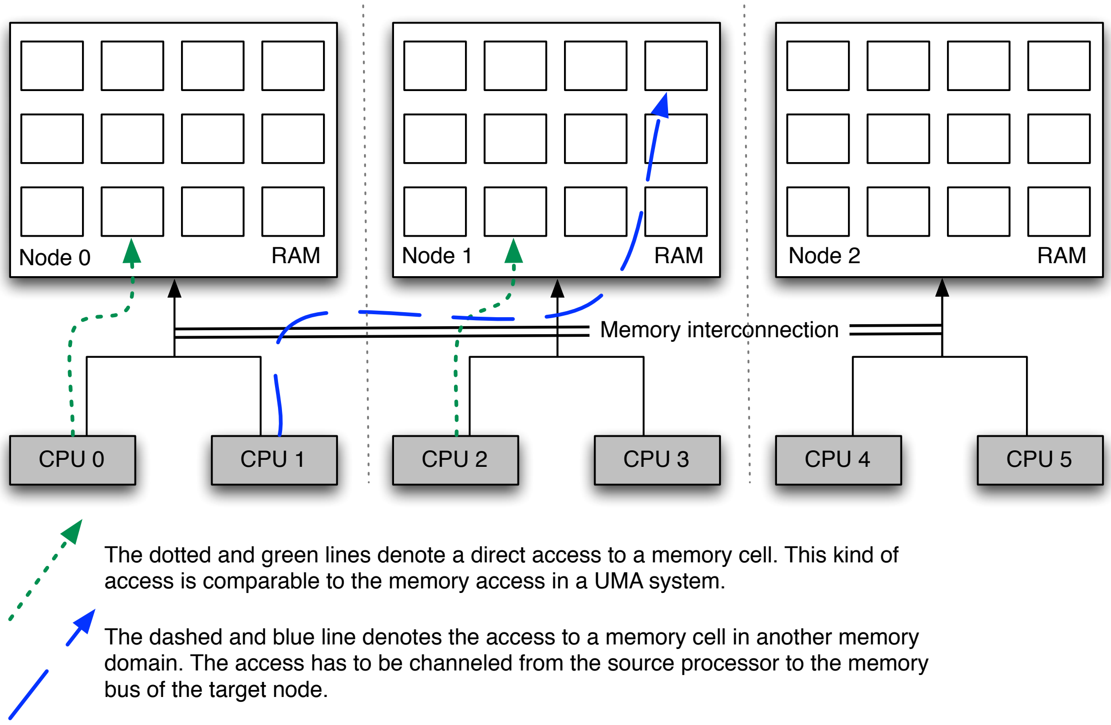
Figure 2.9: *Uniform Memory Access.*
The processors at the bottom access the figure access the memory cells shown at the top of the figure via the memory bus. Each processor can directly access each memory cell. Both the memory bus and the RAM itself can be bottlenecks, as discussed in section 2.4.2.3.

(DDR-SDRAM) memory allows only one row of a DIMM to be accessed simultaneously.

For large systems with many processors and huge amounts of RAM, these are severe disadvantages. The concept of Non Uniform Memory Access is often used in the domain of high performance clusters. Figure 2.10 shows, how NUMA systems partition memory and processors into distinct and autonomous memory nodes. Processors access their local memory, the memory that resides on the same node as the processor, like they would in a UMA system.

If a processor needs to access memory on another node, the access is channelled through the addressed node. Although non-local access is slower than local access, the model has several advantages over the UMA model. If mostly processors from the same node access the memory, the pressure on the memory is reduced because all intra-node memory access on the other nodes is removed from the local memory bus. This greatly eases horizontal scaling by adding more memory nodes to the system (compare to [23]).

Modern x86 multicore systems like AMD Athlon [39] and Intel Nehalem [9]



The dotted and green lines denote a direct access to a memory cell. This kind of access is comparable to the memory access in a UMA system.

The dashed and blue line denotes the access to a memory cell in another memory domain. The access has to be channeled from the source processor to the memory bus of the target node.

Figure 2.10: *Non Uniform Memory Access.*
This NUMA system consists of three memory nodes. Each node is structured comparable to the UMA system in figure 2.9. Processors can directly access the memory in their local memory node. Processors that want to access memory in another memory node must do so by communicating with the memory controller of the addressed node, hence the name *Non Uniform* Memory Access.

also implement a NUMA design. For an introduction into how Linux handles NUMA architectures see [44].

2.4.2.4 Physical Memory Organization

The physical memory on x86 architectures is not set up as homogeneous memory area. The address space of Physical memory is interstratified with various areas of special-treated address ranges. The Basic Input Output System (BIOS), devices connected to the Peripheral Component Interconnect (PCI)- and Industry Standard Architecture (ISA)-bus and others peripherals can reserve memory ranges that are not directly accessible as physical memory.

The most commonly known example for such a special address range is the video memory overlaid at $0xA0000$.

A map that contains all address regions reserved by the BIOS is called e820 map and can be queried from the BIOS [54].

The fragments 2.1 to 2.3 were obtained from a *2.6.34-rc4-gbea481a-dirty* kernel running on an AMD X2 CPU with 8 GiB of RAM and exemplify, how the memory is fragmented. Various other important informations about limited resources are included as well. Noteworthy are the restricted size of the Direct Memory Access (DMA)-zones, that indicate why reserving and testing large memory parts in these regions can lead to a shortage in memory.

A much more detailed instruction into the organization of the physical memory with Linux and 32 bit x86 systems can be found in [62, 89, 46, 53].

2.4.2.5 Memory Organization in User Space

Linux provides each process with a private address space. Depending on the architecture the kernel was compiled for, and the architecture the user space program was written for, the size of the addresses used are 32 or 64 bit [53, 77]. User space programs always use virtual addresss, and are unable to directly read or write a physical address.

To be able to access specific frames is an important precondition for the implementation of a memory tester in user space. Kernel modules can implement an interface that allows user processes to utilize the *mmap* system call to map frames into their address space. The implementation presented in this thesis uses this approach to write a user space based memory tester.

2.4.2.6 Memory Organization in Kernel Space

On 32 bit x86-architectures, the Linux kernel is loaded into the first GiB of physical memory. Virtual addresses pointing to kernel memory start at $PAGE\_OFFSET$,which in x86 is defined as $0xC0000000$ (3 GiB) [26], although this depends on the architecture (x86-32 bit/x86-32 bit + PAE/x86-64 bit/other architectures) and the division of memory between userspace and kernelspace [24]. Section 2.4.2.4 showed, that not all physical memory is usable

```
1  linux2.6.34:~ jens$ dmesg
2  ...
3  BIOS-provided physical RAM map:
4   BIOS-e820: 0000000000000000 - 000000000009fc00 (usable)
5   BIOS-e820: 000000000009fc00 - 00000000000a0000 (reserved)
6  ...
7   BIOS-e820: 0000000100000000 - 0000000230000000 (usable)
8  ...
9  Zone PFN ranges:
10   DMA      0x00000001 -> 0x00001000
11   DMA32    0x00001000 -> 0x00100000
12   Normal   0x00100000 -> 0x00230000
13 ...
14 On node 0 totalpages: 2097006
15   DMA zone: 56 pages used for memmap
16   DMA zone: 0 pages reserved
17   DMA zone: 3942 pages, LIFO batch:0
18   DMA32 zone: 14280 pages used for memmap
19   DMA32 zone: 833544 pages, LIFO batch:31
20   Normal zone: 17024 pages used for memmap
21   Normal zone: 1228160 pages, LIFO batch:31
22 ...
```

Listing 2.1: *Bootmessages*

This is the output the Linux kernel generates at boot time.

Lines 3–7 show the shortened e820 map.

Lines 9–12 show the physical frames (PFN) for the zones *DMA*, *DMA32* and *Normal*. DMA for ISA devices is only allowed in the physical frames $0x00000001 \cdots 0x00001000$, the first 16 Mib, excluding the first 4 KiB. This limitation stems from the ISA specification. *DMA32* is an extended DMA zone for devices capable of accessing it. The initial patch for *DMA32* includes more detailed documentation [78]. *Normal* memory is memory that can directly be accessed by the Linux kernel. 32 bit kernels include a *HIGHMEM* zone (not shown here) that cannot be directly accessed by the Linux kernel, because the virtual address space chosen for the kernel is only 1 GiB big [89]. Compare to figure 2.6, where the zones for Linux running on a 32 bit x86 are shown.

Lines 14–21 show how the zones are filled for each NUMA node. *LIFO batch* is an mm internal configuration that dictates the batch-size used when a zone needs to be refilled.

```
linux2.6.34:~ jens$ cat /proc/cpuinfo
processor : 0
vendor_id : AuthenticAMD
...
cpu cores : 2
...
TLB size  : 1024 4K pages
...
address sizes : 40 bits physical, 48 bits virtual
...
```

Listing 2.2: */dev/cpuinfo*

The information about the installed CPUs includes two items interesting for the mm: The total address size this processor can address, and the size of the TLB.

```
1  linux2.6.34:~ jens$ cat /proc/iomem
2  00000000-00000fff : reserved
3  00001000-0009fbff : System RAM
4  ...
5  00100000-cffcffff : System RAM
6    01000000-014cff4f : Kernel code
7    014cff50-01b77aaf : Kernel data
8    01e0b000-0276fc37 : Kernel bss
9    20000000-23ffffff : GART
10 cffd0000-cffddfff : ACPI Tables
11 cffde000-cfffffff : ACPI Non-volatile Storage
12 d0000000-febfffff : PCI Bus 0000:00
13   d0000000-dfffffff : reserved
14     d0000000-dfffffff : pnp 00:0c
15   e0000000-efffffff : PCI MMCONFIG 0000 [bus 00-ff]
16     e0000000-efffffff : pnp 00:0b
17 ...
18   fd7f9000-fd7f9fff : 0000:00:05.0
19     fd7f9000-fd7f9fff : sata_nv
20 ...
21 fec00000-fec00fff : reserved
22   fec00000-fec003ff : IOAPIC 0
23 ...
24 100000000-22fffffff : System RAM
```

Listing 2.3: */dev/iomem*

This file shows you the current map of the system's memory broken up per device, the complete file on that system contains 57 rows.

Lines 3, 5, and 24 contain the physical addresses that address RAM.

Lines 5–8 show, where the kernel is loaded into the physical memory.

The Graphics Address Remapping Table (GART) in line 9 is used to map the memory of the graphic card into system memory.

ACPI tables (line 10) describe the computer hardware, including the systems address map, power management, and more [54].

by the operating system, and that the memory that is usable contains holes. On 32 bit x86 systems, the physical memory starting after the first MiB ($0x100000$ and onward) is used for the code and data of the Linux kernel [89, chapter 3]. In most cases it is impossible or very difficult to reliably test memory frames used for kernel code and data, as a consequence, the implementation accompanying this diploma thesis does not test for memory errors within kernel memory.

2.4.2.7 Frame Management

Frame management is the fundamental responsibility of the mm. Each frame is associated with a data structure called *struct page*, shown in listing C.1 (page 136) and listing C.2 (page 137).

Each frame is managed by maintaining one *struct page* instance. This leads to 2^{20} distinct *struct page* instances for a frame size of 4 KiB and 4 GiB of managed memory. Consequently a small size of *struct page* instances is desired to reduce the amount of memory needed for frame management. This is achieved by removing members of *struct page* needed for aspects that are not compiled into the kernel, and the polymorphic use of certain memory regions inside a *struct page* instance by using the *C* keyword *union*[4]

As a result, the interpretation of a given *struct page* instance p depends on the subsystem this instance is allocated to, effectively making p an opaque structure that can only be analysed by relying on knowledge that should be encapsulated in the subsystem that owns p. Additionally, the lock hierarchies that protect p or specific attributes of p are partially private to the implementation of the owning subsystem.

An, albeit slightly outdated, discussion of *struct page* can be found in chapter 3 of [89].

2.4.2.8 Frame Management in Zones

The primary management of single frames centers around *struct page*, but not all information available about a frame is stored inside the *struct page* instances.

[4] *union* allows multiple variables to share the same memory space [74].

Frames are assigned to zones. Lines 9–12 in listing 2.1 show exemplary values for the zone-size on a 64-bit Linux kernel. Within the zone, non free frames are kept in the *active/inactive* lists that are used for paging decisions. A limited number of frames are kept for the per-CPU Hot'n'Cold allocator that acts as a small cache for single frame allocations. Mostly, the memory in a zone is managed by the Buddy Allocator described in the next section.

2.4.2.9 Buddy Allocator

Figure 2.6 showed the Buddy Allocator at the bottom of the Linux memory management hierarchy. The Buddy Allocator manages hierarchical pools of ranges of consecutive frames. Each pool P_{order} contains free regions of each 2^{order} free consecutive frames. A client that requests memory does so by requesting 2^n free frames. If the allocator finds a free region in P_n, it is allocated to the client. If no free region can be found in P_n, a region is removed from P_{n+1}, and split in halve into two *buddies*. One of the buddies is allocated to the client, the other is used as new free region in P_n.

Figure 2.11 shows how the Buddy Allocator manages the request for 8 KiB, when only one 32 KiB region is available. A more detailed description of the Buddy Allocator can be found in [108].

The Buddy Allocator in Linux manages memory per zone, and tries to allocate the requested memory on the same NUMA node, the requesting client is running on. In low memory situations, the buddy allocator also utilizes various strategies to free memory. Different strategies are necessary, because e.g. a memory allocation in an interrupt context must be very fast and must not switch the execution context and go to sleep, whereas a user space process has much less stringent requirements. Much of the complexity of the Linux mm lies in the management of free memory. A more detailed description of the Linux implementation of the Buddy Allocator can be found in [89, 46, 62].

2.4.2.10 Relating Frames to Hardware

The Linux kernel has no direct data about which frame is located in which memory module(s). This can be problematic, when the system operator needs

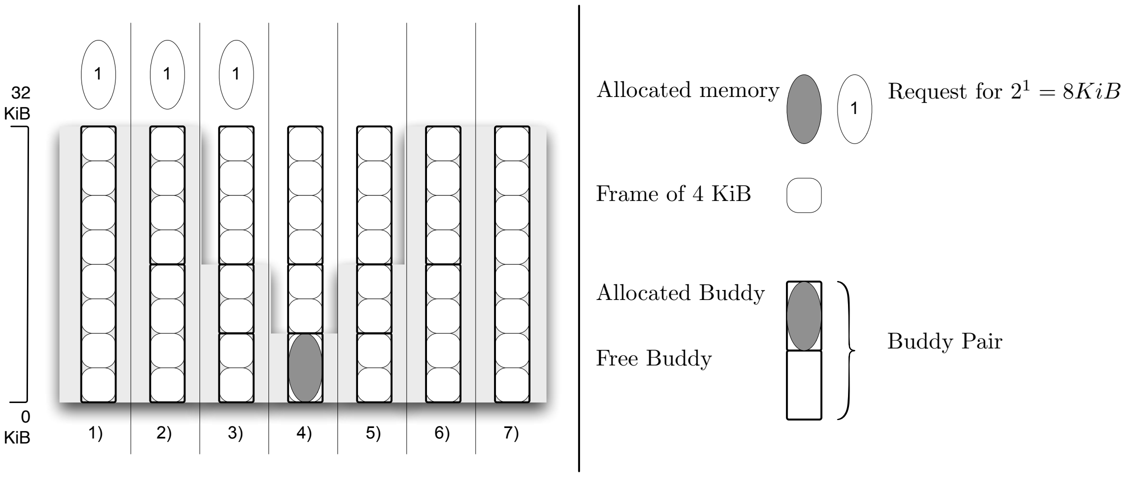


Figure 2.11: *Buddy Allocator.*
This figure shows by example, how the Buddy Allocator splits up a larger buddy to satisfy the request for a smaller buddy.
1) The Buddy Allocator has one free range of 32 KiB (order 3). A request for 8 KiB (order 1) arrives.
2) The buddy is split into halves.
3) The lower halve is recursively split again into two buddies of the order 1.
4) The lower of the order 1 buddies is allocated to the client. The range of the order 3 is now split into two buddies of the order 2, one of them free, the other one again split up into two buddies.
5) The memory region is freed by the client.
6 − 7) Analogous to the recursive decomposition of steps 2+3, the free buddies are recursively merged back.

to find a module with known defects by the physical address Linux uses. Fortunately the BIOS DMI tables mentioned in section 2.4.2.4 often allow physical addresses to be related to memory modules. The program *mcelog*, that will be discussed in the related works chapter, can parse the DMI tables and provide the sought information to the user [30].

2.4.3 Complexity of the Linux Kernel

From its modest beginnings in 1991, the Linux kernel has grown to an impressive size. The following sections illuminate some aspects of the Linux kernel that

```
 1 linux-2.6.34 jens$ git branch
 2 * v2.6.34-rc7
 3
 4 # All non-empty lines that do not contain a comment
 5 linux-2.6.34 jens$ grep -v  ’#’ .config| grep -v ’^$’ | wc -l
 6 3257
 7
 8 # All lines with not set defines
 9 linux-2.6.34 jens$ grep ’not set’ .config|wc -l
10 708
11
12 # Count the number of ifdef/ifndef preprocessor directives in the mm subsystem
13 linux-2.6.34 jens$ grep -R -E ’if[n]?def’ mm|wc -l
14 361
```

Listing 2.4: *Counting the number of configuration options for the Linux Kernel*
The number of configuration options for the Linux kernel can be calculated by analysing the *.config* file [12].
Lines 12–14 extract the number of preprocessor directives embedded in the mm-directory. Only one of these *ifndefs* stems from the common C-header file preamble.

directly influence many of the design decisions that have to be made for an online memory tester.

2.4.3.1 Linux Implementation Aspects

The upcoming kernel 2.6.34 can be configured with nearly 4000 (3257 + 708 = 3965, see listing 2.4) different configuration options. Some of these options exclude each other, and most options determine whether a given device driver should be built or not. Still, some of these options change how certain aspects of the kernel are implemented.
The mm subsystem alone contains 361 conditional preprocessor directives. One of these directives is the preprocessor guard against including the header file multiple times, the other *#ifdef*/*#ifndef*s directly influence the generated kernel.

2.4.3.2 No stable Kernel interfaces

The community of Linux kernel developers has agreed, that kernel internal interfaces and implementation details are not stable and can be changed any time [81]:

> This is being written to try to explain why Linux does not have a binary kernel interface, nor does it have a stable kernel interface. Please realize that this article describes the _in kernel_ interfaces, not the kernel to userspace interfaces. The kernel to userspace interface is the one that application programs use, the syscall interface. That interface is _very_ stable over time, and will not break. I have old programs that were built on a pre 0.9something kernel that still work just fine on the latest 2.6 kernel release. That interface is the one that users and application programmers can count on being stable.

Taking into account, that all books covering the Linux kernel in depth where published between 2000 and 2006 [108, 53, 87, 89, 103, 46, 62], it is no wonder, that all books describe a Linux kernel that differs to a greater or lesser extent from the kernel in 2010.

2.4.3.3 Volatility of Kernel code

The kernel code has gone through a lot of changes during the last years. To visualize these changes, figures 2.12 and 2.13 plot the size and the amount of change the mm subsystem underwent between kernel releases 2.6.11 and 2.6.33. The data has been obtained by analysing the kernel source code in the official Linux-kernel git repository at `git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git`.

2.4.4 How Linux Relates to this Thesis

In section 2.4.1 the choice of Linux as implementation target for the prototype memory tester has been justified. The following sections examine the memory management implementation of the Linux kernel (section 2.4.2) and the complexity and volatility of the kernel (section 2.4.3).
These observations are the base for many of the design decisions made in chapter 4.

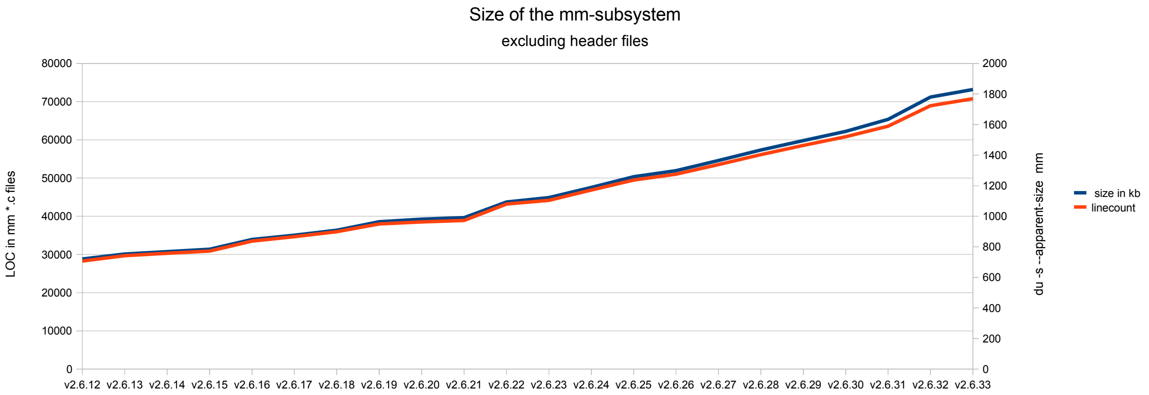


Figure 2.12: *Size of the mm Subsystem*
Both graphs describe the size of mm subsystem. Between 2.6.11 and 2.6.33 the size of the mm subsystem more than doubled.
The left axis measures the size in lines of code, the right axis in KiB of files. Only files in the `mm` directory were counted as mm related code. mm code in other directories is not included in the data.

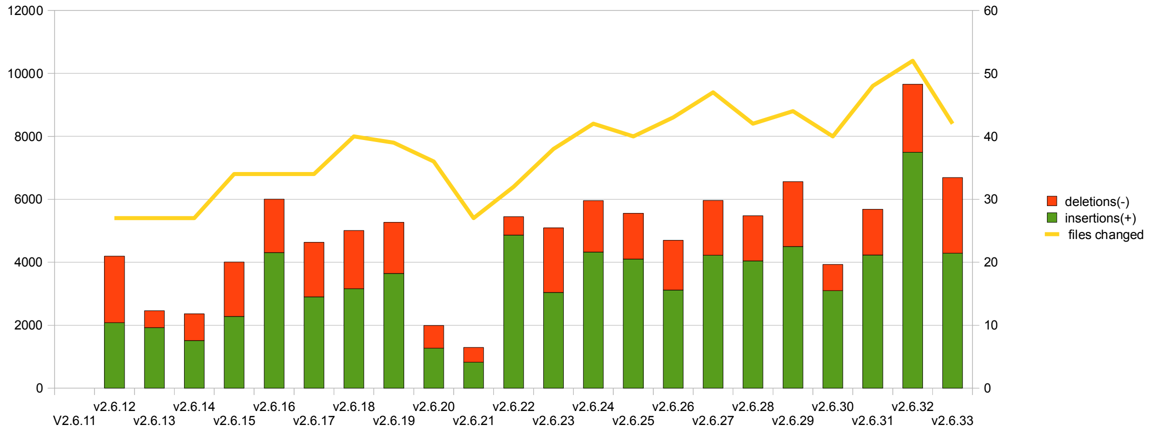


Figure 2.13: *Changes of the mm Subsystem*
Using the same source data as in figure 2.12, this figure describes the changes that happened between the releases.
The left axis shows, from release to release, the number of lines deleted from, or inserted into files in the mm directory.
The right axis shows, how many files were changed in the mm directory.

Chapter 3

Related Work

Detecting and reacting to memory faults in computers has gotten more and more attention in recent years. Improvements in the hardware, for example the extended MCE handling of Intel Nehalem processors [72], lead to improvements in operating systems [9, 113]. Research is not limited to hardware development though, for example other researchers investigate how programming models for software can be extended in a way, that they support fault recovery [92].

This section presents selected work that relates to this diploma thesis. Three different research topics are be presented: memory tests, fault management, and hardware based solutions to detect and correct errors in RAM.

3.1 Memory tests

In section 1.2.2 a classification for memory testers has been introduced. Memory tests were distinguished into *bare metal* memory tests that run directly on the system, without an underlying general purpose operating system, and memory testers running on top of an operating system.

In the following sections, three different memory testers are presented. The first of these is *memtest86+* and is a bare metal tester. The second tester is an exploratory kernel based implementation that had been written for the Solaris 8 kernel. The third tester runs as a user space process on Linux.

These three examples were chosen, because each of them uses increasingly more layers of services to access the physical memory. Discussing the benefits and drawbacks of the respective positions in a software stack (similar to the example given in figure 1.1) is beneficial for the finding of a “sweet spot” for a memory tester.

3.1.1 Memtest86+

memtest86+ [13] is a well known, classical bare metal memory tester, and stands as an example for this class of memory testers. memtest86+ is distributed under the GPL, and widely used. For example it is distributed together with many Linux distributions [6, 7, 8]. A screenshot of memtest86+ is shown in figure 3.1.

The usage of memtest86+ is simple: it is booted from CD or other boot medium, and immediately starts to test for defective memory. Testing will continue, until the user cancels the test procedure. If all test patterns have been verified, the test cycle will restart with the first test pattern. The recommended usage of memtest86+ is, to run it until each test algorithm has been applied to each memory region [14].

Noticeable is that the RAM where the program will finally reside is tested in advance. This allows memtest86 to test all memory accessible by the CPU.

Besides the default settings, several adjustments for advanced users are possible. These settings include to adjust the memory refresh rate and to enable/disable hardware based ECC checks. Adjusting these parameters can cause the memory to be accessed outside its specification, a good instrument to gauge the safety margin of the default parameters the hardware runs with.

The test algorithms implemented by memtest86+ are based on an idealised algorithm (listing 3.1) and are termed *Moving Inversions* and *Modulo-X*. A transcription of the definition found on the memtest86 homepage[14] can be found in listing 3.2 and 3.3 [13].

3.1.1.1 Relation to this Thesis

Bare metal memory testers like memtest86+ share some principal shortcomings and difficulties, on the other hand they have advantages over memory testers that run on top of a general purpose OS.

```
# The first algorithm is, according to the memtest86 description, this
# algorithm detects pattern sensitive faults. It is considered
# to be the ideal algorithm for memory tests.
1. write a cell with a zero
2. write all of the adjacent cells with a one, one or more times
3. check that the first cell still has a zero
```

Listing 3.1: *"Ideal" algorithm for memtest86+*

This pseudo code is the description of the test algorithm that the authors of memtest86 consider to be "an ideal strategy for testing memory" [14].

The pseudo code has been copied from [14].

```
1. Fill memory with a pattern

2. Starting at the lowest address
 2.1 check that the pattern has not changed
 2.2 write the patterns complement
 2.3 increment the address
 2.4 repeat

3. Starting at the highest address
 3.1  check that the pattern has not changed
 3.2  write the patterns complement
 3.3  decrement the address
 3.4  repeat
```

Listing 3.2: *Moving Inversions algorithm used by memtest86+*

This pseudo code is the description of memtest86+'s *Moving Inversions* test.

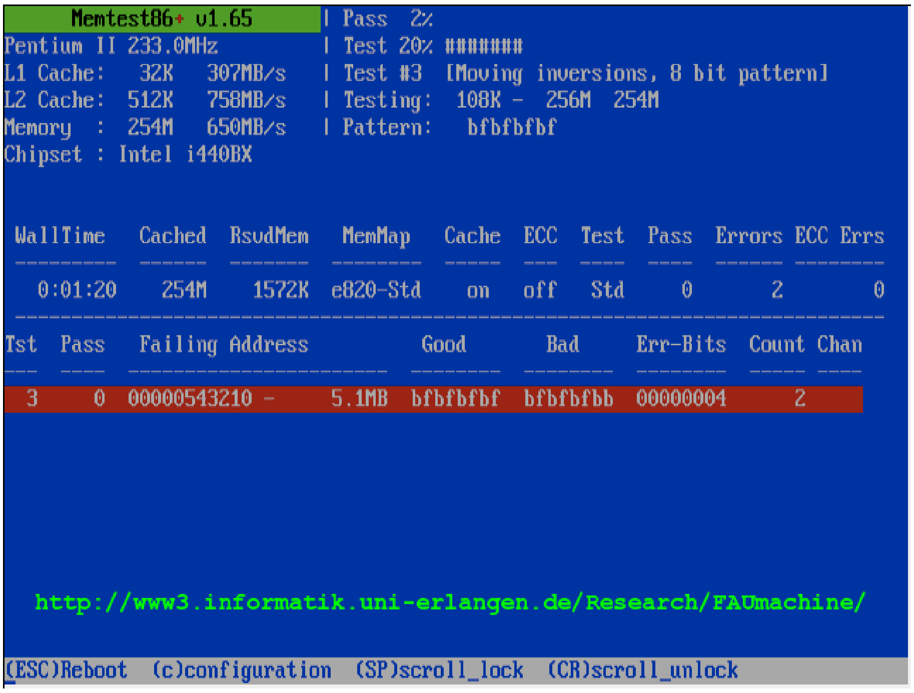
The pseudo code has been copied from [14].

```
1. For starting offsets of 0 - 20 do
   1.1 write every 20th location with a pattern
   1.2 write all other locations with the patterns complement
   1.3 repeat above one or more times
   1.4 check every 20th location for the pattern
```

Listing 3.3: *Modulo-X algorithm used by memtest86+*

This pseudo code is the description of memtest86+'s *Modulo-X* test.

According to the author of Memtest86 (the predecessor of memtest86+), the *Modulo-X* algorithm is not affected by caching or buffering. The pseudo code has been copied from [14].



```
     Memtest86+ v1.65      | Pass  2%
Pentium II 233.0MHz        | Test 20% #######
L1 Cache:   32K   307MB/s  | Test #3  [Moving inversions, 8 bit pattern]
L2 Cache:  512K   758MB/s  | Testing:  108K -  256M  254M
Memory  :  254M   650MB/s  | Pattern:   bfbfbfbf
Chipset : Intel i440BX


 WallTime   Cached  RsvdMem   MemMap   Cache  ECC  Test  Pass  Errors ECC Errs
 ---------  ------  -------  --------  -----  ---  ----  ----  ------ --------
   0:01:20    254M    1572K  e820-Std    on   off   Std     0       2        0
 -----------------------------------------------------------------------------
Tst  Pass   Failing Address          Good       Bad     Err-Bits  Count Chan
---  ----  -----------------------  --------  --------  --------  ----- ----
  3     0  00000543210 -     5.1MB  bfbfbfbf  bfbfbfbb  00000004      2






   http://www3.informatik.uni-erlangen.de/Research/FAUmachine/

(ESC)Reboot  (c)configuration  (SP)scroll_lock  (CR)scroll_unlock
```

Figure 3.1: *Screenshot of memtest86x*
This image is a screenshot of the bare-metal memory tester *memtest86+*, and shows a detected memory fault.
The screenshot has been obtained from `http://www3.informatik.uni-erlangen.de/Research/FAUmachine/`.

The first and most noticeable shortcoming of a bare metal memory tester is that it runs as the sole program on the computer. From a user's perspective, this is a disadvantage, because it causes a downtime of the services this computer provides. Though unwanted by te user, this monopolisation of the hardware allows the test program to access the memory without regard to anyone else. By disabling interrupts and DMA, a bare metal tester can increase its accuracy by eliminating external memory accesses. Additionally, the memory tester can modify hardware parameters that could have adverse side effects on the stability of the system.

A program that runs without an underlying operating system can have full hardware access, the advantages of this have been described in the previous paragraph. The drawback of disregarding the services of an operating system is that the program cannot rely on the hardware abstractions provided by the operating system. In the case of a memory tester this means, that many hardware near functionalities must be implemented. Detecting the memory regions explored in section 2.4.2.4 can be a problem, if for example the memory map provided by the BIOS is erroneous. Operating systems like Linux provide corrections or workarounds for many of these issues.

To be able to modify hardware parameters, the memory test has to include drivers for different types of hardware. Other challenges are discovering NUMA nodes and their respective local memory, and to utilize multiple processors to test for memory errors.

Regarding the goals of this thesis, bare metal memory testers like memtest86+ can be considered as complementary to memory testers that run on top of an operating system. The later run under a more constrained environment, because several of the options open to bare metal scanners cannot be used. On the other hand, memory tests running in parallel to the tasks the computer is intended to do from the users point of view, can permanently test for defective memory.

Both types of memory test are complementary, if bare metal testers are rated as hardware specific tools that are used when the memory is already suspected of being faulty, and OS based memory testers are understood as a permanent monitoring and probing tools that assess the reliability of the installed RAM without disturbing normal usage of the computer.

3.1.2 Online Test for Solaris 8

An example for a memory test running on top of an operating system is the implementation Singh, Bose and Darisala from Sun Microsystems presented in their 2005 publication “Software Based In-System Memory Test For Highly Available Systems” [109].

Singh *et al.* implemented a memory tester that is comparable to the approach taken in this diploma thesis – a closer distinction between the two approaches follows in section 3.1.2.1. Implementation wise, they chose a simple, kernel based architecture that sequentially tested memory ranges by applying the fault model of Nair *et al.* (section 2.2.3). The implementation of the frame scheduler abstraction introduced in section 1.2.2.1 is very minimalistic and calculates the to-be-tested memory range based on the configured allocation size and the test-iteration number as $Start\ Address = (Physical\ Memory\ Start\ Address \times Iteration\ Number)$. The fault management steps proposed by Singh *et al.* are a small subset of the fault management process described in section 2.3.4. In detail they proposed, that frames with defects should be excluded from further usage and that processes that accessed a frame at the time of an UE should be killed.

The experimental results of Singh's research are interesting. They installed faulty memory modules into a server and ran, as two different experiments, two different workloads. The faults detected in the respective experiments differed between the experiments. Only about a third (30.4% respectively 27.3%) of the detected errors overlapped – a result that correlates with findings of Schroeder *et al.* in 2009 [105]. One of the results of this study is 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.

3.1.2.1 Relation to this Thesis

Superficially, the presented work of Singh *et al.* resembles the research of this diploma thesis. However, there are profound differences between both works. Singh *et al.* did not publish any performance related results – an important information, if the feasibility of online memory tests needs to be evaluated. Further, neither source code nor a detailed description of the implementation is presented – another important element that is needed for research that bases on this implementation. Other differences are the mode of implementation: Kernel based (Singh) versus a mixture of user- and kernel based (this thesis), and the targeted hardware (SPARC vs. x86) and operating system (Solaris 8 vs. Linux).

In a private communication with the author of this diploma thesis, Singh answered to the question, whether further research has been undertaken by his group, that "I am not aware of any other papers.". Further investigations regarding additional information about this study did not find additional research either.

3.1.3 Effo GPL

Complementary to the work of Singh *et al.*, a group of developers released a – disregarding a minuscule kernel module to translate a virtual address into a physical address – purely user space based implementation of a memory tester. The memory tester targeted the Linux operating system and implemented several of the fault models implemented by memtest86+ [28].

The problem of frame acquiration had been solved in very straight forward manner: The program allocated all the memory it could get and used said kernel module to translate virtual addresss to physical addresss. By iterating over each page in the allocation, the corresponding frame could be identified. Although this proceeding is not described *expressis verbis*, it can be deducted by analysing the source code.

Unfortunately the publication that described the program and test algorithms have been removed from the programs website.

Other implementations of purely user space memory tests exists, for example the *memtester* program [29]. These programs often work similar to the Effo implementation: They allocate memory and test it, often by *mmap*ing */dev/mem* and execute memory tests on the mapped memory.

3.1.3.1 Relation to this Thesis

Compared to the memory tester implemented by Singh *et al.* and memtest86+, the Effo implementation brings together some of the strengths and some of the weaknesses of both approaches.

Similar to a bare metal memory test, the Effo implementation monopolizes the system memory. From user space the implementation cannot directly acquire *specific* frames, it can only do so by allocating many frames and hope, that the sought frame is included there. Allocating all available memory is an effective, albeit not very efficient, workaround to this problem. By reducing the amount of memory allocated, the pressure on the memory system can be reduced, although this goes at the cost that the probability that the sought frames are included in the allocated memory is reduced. 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. In contrast to a bare metal memory test, this implementation cannot directly influence hardware parameters or access all available memory, due to fact that the kernel requires private memory as well. Similar to the approach Singh *et al.* took, the Effo implementation utilized the operating systems capabilities to abstract from specific hardware details. Additionally, the downtime of the system can be reduced to a time of degraded performance.

This user space implementation shows promise to be a usable implementation of a user space memory test. What this implementation is lacking, is the possibility to allocate specific frames in a direct, non performance detrimental fashion. The architecture designed in this diploma thesis not only takes the existence of third party memory tests into consideration, it specifically tries to enforce Dijkstras idea of the *separation of concerns* [58] and treats specific memory test implementations as replaceable components.

3.2 Fault Management

Section 2.3 described an idealised fault management architecture. This section presents three different approaches to fault handling. Two of them, the Linux EDAC project [4] and the Linux mcelog project [30] are narrowly focused on the aspects of sensing memory faults detected by the memory hardware and to react to these faults.

The other project is the *Predictive Self Healing* technology for Sun (now Oracle) Solaris 10 and OpenSolaris, and is a complete implementation of a fault management architecture [20].

3.2.1 Solaris Predictive Self-Healing

In late 2004, the Sun employee Michael Shapiro published an article titled “Self-Healing in Modern Operating Systems” in the ACM's periodical *Queue* [107]. He argued that computer systems in the year 2004 are still far less reliable, than it is desirable. Improving reliability, he wrote, can be done by doing three things: By improving the quality of each component, by introducing redundancies to cope with failures, and by predictively avoiding failure, or at least to reduce the downtime caused by unexpected failures.

To predict imminent failures, warning signals must be identified, and analysed. According to Shapiro, the task of fault prediction falls to a *fault manager* that interprets hardware errors and signals and semantically enriches them. If a hardware component is diagnosed to be broken, or expected to fail in the near future, the fault manager tries to offline the component in a controlled manner. To reduce the downtime caused by interruption of hardware availability – be it expected failures like the controlled offlining of a component, or an unexpected fault – a service management comparable to the one described in section 2.3 is used.

Predictive Self Healing has been introduced in the "Solaris Express 6/04"-release and been enhanced in subsequent releases [20].

This technology implements many of the aspects examined in section 2.3 and, as will be discussed later, influenced the design of the FMA presented in this thesis.

To understand, what the developers of Solaris 10 understood under the term *Predictive Self Healing* it is insightful to quote an excerpt from the "Writing Device Drivers" book from Sun [22, chapter 13]:

> A system like the Solaris OS predictive self-healing system is first and foremost self-diagnosing. Self-diagnosing means the system provides technology to automatically diagnose problems from observed symptoms, and the results of the diagnosis can then be used to trigger automated response and recovery.

The following sections give a high level overview of several techniques used by Solaris Predictive Self-Healing.

3.2.1.1 Service Management

Analogous to the architecture presented in section 2.3, the implementation of fault management in Solaris relies on a service management implementation to start, stop, and restart services. A detailed description of the Solaris service management facility (SMF) can be found in [104, 57].

Contrary to the *init.d* start/stop scripts historically used on UNIX systems, services registered in the SMF are annotated with meta data that declares, besides others, the services Fault Management Resource Identifier (FMRI), the dependencies of the service, the location of the start- and stop-scripts, and configuration parameter private to the specific service instance. Listings 3.4 to 3.6 give an overview of a services properties and management.

3.2.1.2 Enriched Fault Logs

An important step towards automated fault management is the movement from unstructured, and unconnected log messages to standardized, structured, and semantically enriched messages.

Operating systems and their accompanying software often log errors as text messages addressed to the system administrator. In the case of a (suspected) fault, the system administrator had to read the logfiles, and diagnose the fault that happened, trying to deduce the impact of the fault and the required corrective action. Faults that, as the software stack in figure 1.1 exemplified, are the results of other faults are notoriously difficult to analyse by reading logfiles. The administrator has to correlate the content of logfile entries, to detect the whole chain of events that lead to the observed fault, which is often the outage of services provided by the system.

Standardizing error messages can help to analyse these messages automatically. A common protocol describes common aspects of error-, or fault events, and an encoding of an event. The different subsystems define event classes, e.g. *ereport.cpu.amd.nb.mem_ce* for CEs on AMD platforms. Resources, e.g. services and hardware components, are each identified by an FMRI. An example of the FMRIs assigned to services is shown in listing 3.4.

Events are observed by the *Fault Manager*-daemon ($fmd(1M)$). This daemon implements functionalities comparable to the *Diagnosis* and *Response* components of the FMA presented in section 2.3. Listing 3.7 shows the case of a faulted and repaired PCIe device.

```
 1 # List all services registered with the SMF
 2 jens@xvm:~$ svcs -a
 3
 4 STATE          STIME    FMRI
 5 legacy_run     May_17   lrc:/etc/rc2_d/S20sysetup
 6 ...
 7 online         May_17   svc:/system/filesystem/minimal:default
 8 ...
 9 disabled       May_17   svc:/network/talk:default
10 ...
11 online         May_17   svc:/network/nfs/server:default
12 online         May_17   svc:/milestone/multi-user-server:default
13 ...
14 maintenance    May_17   svc:/milestone/xvm:default
```

Listing 3.4: *Using the Solaris SMF: Listing Services*

Listings 3.4 to 3.6 highlight aspects of the SMF. This listing presents the output of the *svcs* command that lists all managed services.

Lines 5–14 show an excerpt of the services managed by the SMF on a computer running OpenSolaris, the total number of services managed on this instance is 220.

Line 5 contains a service that is managed by *init.d*-style run-scripts, and not directly by the SMF. Although the SMF does not directly managed these services, it tracks theirs existence for completeness.

Line 9 shows a service that has been disabled, i.e. the service will not start automatically.

Line 11 shows a service that has been started by the SMF, and that is currently running. The SMF will restart the service if it faults.

SMF used the annotated dependency information presented in listing 3.4 to replace the UNIX *runlevels* by dependency-only services called *milestones*, e.g. lines 12 and 14.

Line 14 contains a service that is in maintenance mode, i.e. the service has faulted too often and is disabled until the operator tells the SMF, that the service is repaired.

```
 1 jens@xvm:~$ uname -a
 2 SunOS xvm 5.11 snv_133 i86pc i386 i86xpv Solaris
 3
 4 # Show details for the xvm service
 5 jens@xvm:~$ svcs  -l svc:/milestone/xvm:default
 6 fmri         svc:/milestone/xvm:default
 7 name         xvm milestone
 8 enabled      true
 9 state        maintenance
10 next_state   none
11 state_time   May 17, 2010 01:09:21 PM CEST
12 logfile      /var/svc/log/milestone-xvm:default.log
13 restarter    svc:/system/svc/restarter:default
14 dependency   optional_all/none svc:/system/xvm/store (online)
15 dependency   optional_all/none svc:/system/xvm/console (online)
16 dependency   optional_all/none svc:/system/xvm/domains (online)
17 dependency   optional_all/none svc:/system/xvm/virtd (online)
18 dependency   optional_all/none svc:/system/xvm/xend (online)
19
20 # Show the configuration of the xvm service
21 jens@xvm:~$ svcprop  svc:/milestone/xvm:default
22 general/enabled boolean true
23 general/entity_stability astring Unstable
24 general/single_instance boolean true
25 hypervisor/pin_vcpus boolean false
26 ...
27 restarter/logfile astring /var/svc/log/milestone-xvm:default.log
28 restarter/start_pid count 2505
29 restarter/start_method_timestamp time 1274094561.653359000
30 restarter/start_method_waitstatus integer 24320
31 restarter/transient_contract count
32 restarter/auxiliary_state astring method_failed
33 ...
```

Listing 3.5: *Using the Solaris SMF: Service Configuration*

This excerpt from a shell session shows how the Solaris SMF configures services.

Services managed by the SMF are treated as distinct objects that can be managed. Lines 6–18 show several properties of the *xvm* ([36]) milestone. For example, line 13 names the start/stop implementation to be used, lines 14–18 list the services the milestone depends on.

A more detailed view of the milestones configuration is shown in lines 22–33 which show how the configuration of the service is stored in a unified, structured storage system.

```
# Tell the SMF, that the xvm service is no longer
# in maintenance.
jens@xvm:~$ svcadm clear svc:/milestone/xvm:default
# The SMF will now try to restart the service
# ..
```

Listing 3.6: *Using the Solaris SMF: Service Management*

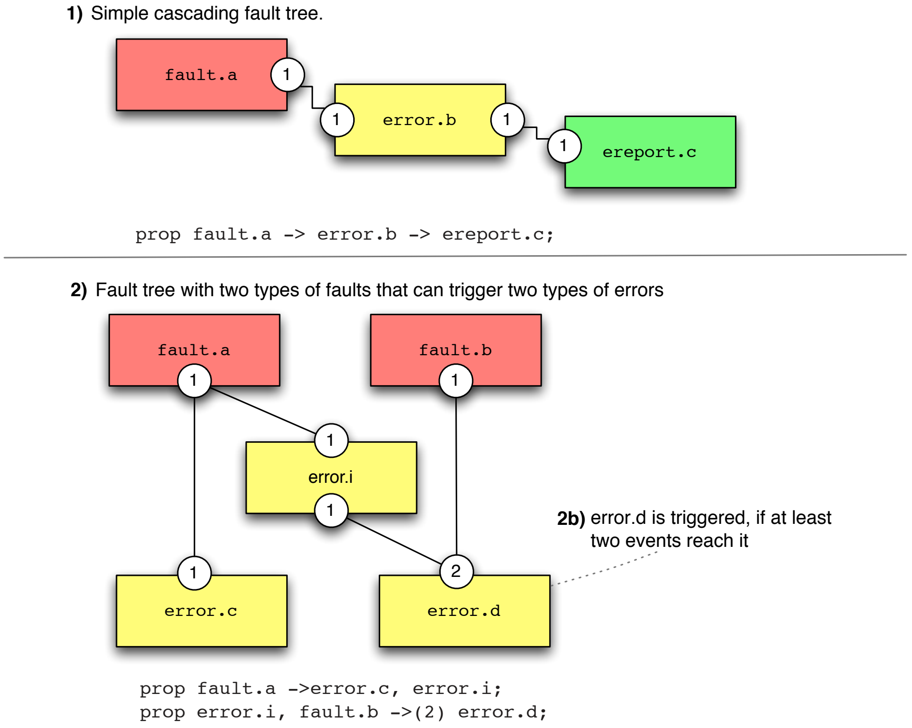
This excerpt is an example for service management: The command *svcadm clear* $\cdots/xvm\cdots$ tells the SMF, that the xVM service has been repaired, and should be cleared of its *maintenance* state.

```
1  jens@xvm:~$ fmdump
2  TIME                 UUID                                 SUNW-MSG-ID
3  ...
4  Feb 27 21:19:30.2936 8e9ea33f-9c2f-4ee9-ce31-9ba0b861af5f  PCIEX-8000-J5
5  ...
6  Feb 27 21:50:04.7108 8e9ea33f-9c2f-4ee9-ce31-9ba0b861af5f  FMD-8000-4M Repaired
7  Feb 27 21:50:04.7275 8e9ea33f-9c2f-4ee9-ce31-9ba0b861af5f  FMD-8000-6U Resolved
8  ...
9  jens@xvm:~$ fmdump -v -u 8e9ea33f-9c2f-4ee9-ce31-9ba0b861af5f
10 TIME                 UUID                                 SUNW-MSG-ID
11 Feb 27 21:19:30.2936 8e9ea33f-9c2f-4ee9-ce31-9ba0b861af5f  PCIEX-8000-J5
12   100%  fault.io.pciex.device-interr-corr
13
14         Problem in: hc://:product-id=PowerEdge-T300:server-id=xvm:chassis-id=G46WD4J/motherboard=0/
              hostbridge=7/pciexrc=7/pciexbus=1/pciexdev=0/pciexfn=0
15            Affects: dev:////pci@0,0/pci8086,2948@1c,4/pci1028,210@0
16                FRU: hc://:product-id=PowerEdge-T300:server-id=xvm:chassis-id=G46WD4J/motherboard=0
17           Location: MB
18
19 Feb 27 21:50:04.7108 8e9ea33f-9c2f-4ee9-ce31-9ba0b861af5f  FMD-8000-4M Repaired
20   100%  fault.io.pciex.device-interr-corr       Repair Attempted
21
22         Problem in: hc://:product-id=PowerEdge-T300:server-id=xvm:chassis-id=G46WD4J/motherboard=0/
              hostbridge=7/pciexrc=7/pciexbus=1/pciexdev=0/pciexfn=0
23            Affects: dev:////pci@0,0/pci8086,2948@1c,4/pci1028,210@0
24                FRU: hc://:product-id=PowerEdge-T300:server-id=xvm:chassis-id=G46WD4J/motherboard=0
25           Location: MB
26
27 Feb 27 21:50:04.7275 8e9ea33f-9c2f-4ee9-ce31-9ba0b861af5f  FMD-8000-6U Resolved
28   100%  fault.io.pciex.device-interr-corr       Repair Attempted
29
30         Problem in: hc://:product-id=PowerEdge-T300:server-id=xvm:chassis-id=G46WD4J/motherboard=0/
              hostbridge=7/pciexrc=7/pciexbus=1/pciexdev=0/pciexfn=0
31            Affects: dev:////pci@0,0/pci8086,2948@1c,4/pci1028,210@0
32                FRU: hc://:product-id=PowerEdge-T300:server-id=xvm:chassis-id=G46WD4J/motherboard=0
33           Location: MB
```

Listing 3.7: *Using the Solaris SMF: Fault Management*

Once faults have been recorded by the fault management, they are grouped in to *cases*. Each case is identified by unique identifier (UUID).

Lines 1–7 show the history of the case with the uuid $8e9 \cdots af5f$. Three events are assigned to this case: A fault, an attempt to repair the device, and the successful reparation of the device. Lines 9–33 cover these events in more detail. The problem location (lines 14, 22, and 30) is a path that uniquely identifies the faulted device and functionality.


1) Simple cascading fault tree.
fault.a
1
1
error.b
1
1
ereport.c
2) Fault tree with two types of faults that can trigger two types of errors
fault.a
1
fault.b
1
1
error.i
1
2b) error.d is triggered, if at least two events reach it
1
error.c
2
error.d

```
prop fault.a -> error.b -> ereport.c;
```

```
prop fault.a ->error.c, error.i;
prop error.i, fault.b ->(2) error.d;
```

Figure 3.2: *Eversholt Fault Trees*
These examples are taken from the book *Eversholt Fault Tree Language* ([114]) and show two simple fault trees. The placeholders *fault.a*, *error.b*, etc. stand for fault and error events that can be triggered by components of the system.

3.2.1.3 Predicting Faults

In simple cases, fault prediction can be implemented as programs written in a general purpose language. Although using general purpose languages is possible, a domain specific language and framework can greatly ease this task. Solaris features the Eversholt language to describe fault trees [114].

A fault tree describes, how faults propagate through a system. A simple example would be a tree `prop fault.a -> error.b; prop error.b -> ereport.c;`. Figure 3.2 shows two fictional fault trees from the Eversholt manual [114].

The placeholder events in the figure can be annotated with constraints, e.g. “`fault.cpu.ultrasparcIII.overtemp@sb/cpu,FITrate=20;`”, so complex fault trees can be specified with relative ease.

Diagnosing specific faults is assisted by engines that act as filters for faults. For

```
/* Example SERD engine declaration  that triggers a report
* if too many correctable errors have been found by the CPU.
*
* In this example, "too many" is 10 CE-events in 12 hours.
*/
engine serd.cpu.ultrasparcIII.ce@sb/cpu,
  N=10,
  T=12 hours,
  trip=ereport.cpu.ultrasparcIII.too_many_ce@sb/cpu;
```

Listing 3.8: *Example of the Eversholt SERD engine*

> “The SERD algorithm is a thresholding algorithm that uses the above properties [N, T, and *trip*] to decide when to *trip* and issue the specified event. The properties N and T describe the threshold as a density of events in time (N events within time T). The *trip* property specifies which event to issue when the Soft Error Rate Discrimination (SERD) engine detects that the events exceed the threshold.” [114]

example, to distinguish soft- from hard-errors, a SERD engine can be utilized. See listing 3.8 for an example taken from the Eversholt manual.

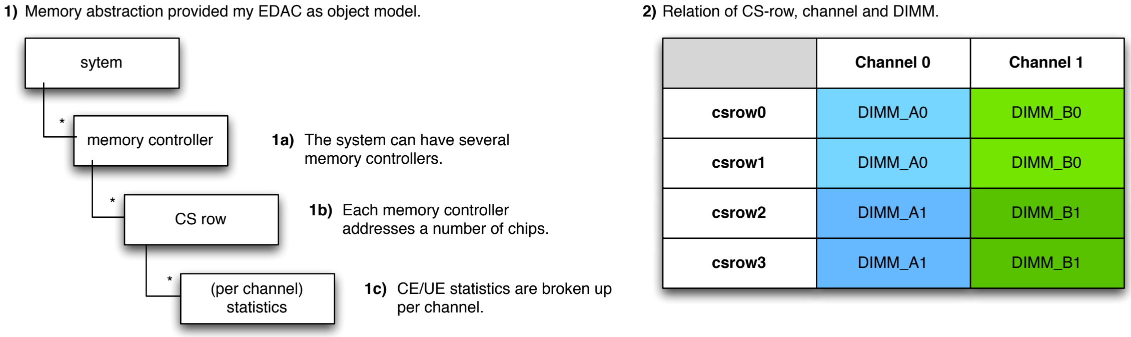
3.2.1.4 Relation to this Thesis

The theme comlex surrounding FMA and SMF is very complex and requires many parties to agree on a common approach. In contrast to Solaris, where kernel and userland origin in the same source, Linux is only the kernel. The userland is specific to the distribution, making a unified approach at service- and fault handling very difficult to coordinate. The website *distrowatch* lists 11 major Linux distributions and more than a hundred lesser known distributions [33], plainly showing the diversity of Linux distributions.

The Solaris *Predictive Self Healing* have been of great influence to the Fault Management Architecture described in section 2.3 and many of the ideas found in the Solaris implementation were used in the presented abstraction. From a software engineers viewpoint, Solaris *Predictive Self Healing* is an *implementation* of an Fault Management *Architecture*. A FMA implementation for Linux would be well advised to learn and adapt its FMA solution(s) from Solaris.

3.2.2 The Linux EDAC project

Moving from the bigger picture of a complete FMA-implementation to fault detection for RAM, this section presents the Linux EDAC patches [4]. EDAC stands for *Error Detection And Correction*, and is the name of several patches and kernel modules that help to handle hardware-related errors, there focusing on ECC errors in memory. As of kernel version $2.6.27-rc5$, the EDAC patches have been merged with the Linux vanilla kernel.



1) Memory abstraction provided my EDAC as object model.

1a) The system can have several memory controllers.

1b) Each memory controller addresses a number of chips.

1c) CE/UE statistics are broken up per channel.

2) Relation of CS-row, channel and DIMM.

	Channel 0	**Channel 1**
csrow0	DIMM_A0	DIMM_B0
csrow1	DIMM_A0	DIMM_B0
csrow2	DIMM_A1	DIMM_B1
csrow3	DIMM_A1	DIMM_B1

Figure 3.3: *Memory abstraction provided by EDAC.*
This figure presents the memory layout, as it is presented to the user space by the EDAC patches [117].
1) corresponds to the object hierarchy EDAC builds in the `/sys/devices/system/edac` directory.
2) shows, how four DIMMS (DIMM_A0, DIMM_A1, DIMM_B0, DIMM_B1) on two channels map to the *csrow* directories in **1b)** .

The EDAC patches set up a hardware abstraction mechanism with two main objectives. On objective is to provide an abstract layer that represents the physical layout of memory in modules, down to the level of the modules Chip Select (CS) rows. Figure 3.3, points 1a-c) show this object model. This requires dedicated hardware drivers for specific chipsets, and memory controllers.

The second objective is to detect ECC-errors in non-RAM components, such as buses, DMA engines, $L1\cdots3$ caches, etc [117]. EDAC is not CPU architecture specific, and currently, there are drivers for both x86 and ppc architectures.

3.2.2.1 Relation to this Thesis

A memory tester running on a machines provokes errors in the memory subsystem. A fault management implementation handles the effects of detected faults. Tools like the EDAC patches connect both systems by making detected memory faults – that have probably been provoked by the memory tester – accessible to the fault management implementation. By relating memory faults to DIMM modules, the EDAC patches ease the system administrators task of replacing defective modules.

3.2.3 mcelog

Current Linux kernels like version 2.6.34 log machine check exceptions into */dev/mcelog*. This includes exceptions that report CEs and UEs. A program called *mcelog* reads and parses these exceptions, and allows actions to be triggered, when errors are found [30].
To quote from the *README* file that accompanies the *mcelog* package:

> mcelog is the user space backend for logging machine check errors reported by the hardware to the kernel. The kernel does the immediate actions (like killing processes etc.) and mcelog decodes the logfiles.

The *mcelog* program provides several ways, in which it can react to MCE-events: Polling, triggering by the kernel, and running as a daemon. When *mcelog* is run as daemon, it tries to predict hardware errors.
mcelog rivals with EDAC, and this is one of the reasons, why it is included here.

3.2.3.1 Relation to this Thesis

In functionality, *mcelog* is comparable to EDAC, although the level of detail reported by *mcelog* is lower, and EDAC does not feature fault prediction . At the time of writing this diploma thesis, a “Hardware Error Kernel Subsystem Mini-Summit” [31] has been held, the protocol can be found at [32]. The goal of the summit was to plan a unified strategy for fault analysis and handling within the Linux kernel. Although the author of *mcelog* was not attending the summit, he participated in the discussion following the protocol [32]. The discussion following the publication of the protocol made clear, that fault reporting and handling is a very controversial subject, and that further development in this area is to be expected.

Regarding this diploma thesis, both EDAC and *mcelog* are potential components both for ECC fault detection, and as (to be) established standard entry point into a fault management implementation.

3.3 Hardware Based

Section 1.2.1 motivated the usage of hardware based error detection and correction schemes. The following two sections introduce ChipKill, an improvement of ECC, and recent research about *virtualized* ECC, i.e. modifying the memory subsystem to allow the operating system to implement the ECC functionality in software.

3.3.1 Chipkill

Memory subsystems equipped with ECC error correction are a great improvement over memory subsystems without any detection or correction abilities – this has already been discussed. In 1997 Timothy Dell published a whitepaper that discusses shortcomings of SEC-DED ECC [56]. Dell argues, backed by field research done by IBM and statistical simulation, that errors that affect multiple bits are very likely and are not handled by SEC-DED codes. He further argued that hard errors that affect multiple bits (multibit hard errors) are often caused by a single memory chip that fails completely, figure 3.4 shows an example of this.

The failure of a single of these chips affects more than one bit, a situation that SEC-DED ECC codes cannot handle. A ChipKill protection scheme can withstand the failure of a single memory chip. This can be accomplished by multiple means, see for example [51, 99, 119] and [56]. On possible solution to the problem of multi-bit correction with standard ECC modules is to expand

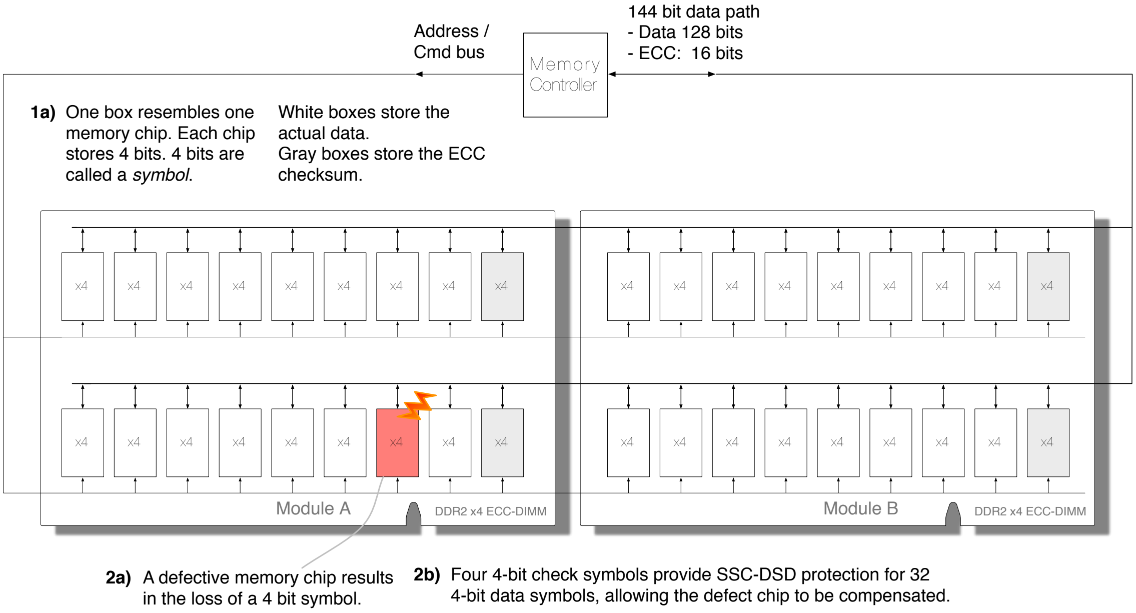

144 bit data path
- Data 128 bits
- ECC: 16 bits
Address /
Cmd bus
Memory
Controller
1a) One box resembles one memory chip. Each chip stores 4 bits. 4 bits are called a *symbol*.
White boxes store the actual data.
Gray boxes store the ECC checksum.
x4
Module A
DDR2 x4 ECC-DIMM
Module B
DDR2 x4 ECC-DIMM
2a) A defective memory chip results in the loss of a 4 bit symbol.
2b) Four 4-bit check symbols provide SSC-DSD protection for 32 4-bit data symbols, allowing the defect chip to be compensated.

Figure 3.4: *DDR memory module.*
DDR modules are build from multiple memory chips. The address width of these chips is 4, 8 or 16 bits. Depending on the type of chips used to build a DDR memory module, the module is called a $x4$, $x8$, or $x16$ module.
This figure shows two $x4$ modules, combined in a way used by Sun [112] and AMD [40] to implement ChipKill. In this example, one of the chips (**2a)**) is defective. In a regular SEC-DED -setup, this would lead to un*detectable* memory errors, because 4 out of 64 bits were faulty. By calculating the checksum over the combined data path of 128 bits, the missing symbol can be reconstructed (see [119], where this image has been adapted from, for more details).

the granularity of the memory access. Instead of protecting each 64+8 bit word separately, multiple words are protected together. To protect against the failure of a $x4$ chip, four $64 + 8$ bit words are protected together $(128 + 16 = 144)$. Although this lowers the access granularity and forces the memory subsystem to read and write a minimum of 144 bits (instead of 72 bits), this scheme can protect against the failure of a complete memory chip. Sun UltraSPARC-T1/T2 [112] and AMD Opteron [40] systems implement this strategy by accessing two modules simultaneously.

3.3.1.1 Relation to this Thesis

Memory protected by ChipKill is more resilient to errors that memory protected by ECC [56]. This puts systems equipped with ChipKill technology out of the scope of this diploma thesis. Still, ChipKill is an interesting technology that helps to detect and prevent memory errors by using specialised hardware.

The next section presents an approach that virtualizes the ECC subsystem and makes it accessible to the operating system. Yoon and Erez, the authors of the research presented in the next section, implement ChipKill on their system.

3.3.2 Virtualized ECC

Although the focus of this diploma thesis is the detection of memory errors by running software based memory tests, the gap between software based memory tests and hardware based error detection narrows. In March 2010 Yoon and Erez from the University of Texas at Austin presented their seminal paper *Virtualized and flexible ECC for main memory* [119]. All benchmarks and evaluations presented by the authors of said paper are based on simulating the modified memory subsystem.

The idea behind their research was to involve the operating system in the process of fault detection and correction. To do so, they allowed the operating system to participate in the checksumming process by giving it influence on if, and where the checksumming data for a memory word is to be stored. Instead of storing the ECC checksums in the dedicated memory chips on the modules, the operating system was free to store the checksums somewhere in physical memory. This then, allowed the implementation of ECC and ChipKill protection with, and without installed ECC-modules.

A further refinement of the virtualized ECC mechanism is, that the operating system can choose the desired level of protection for memory regions, e.g. the operating system could choose to heavily protect memory used by the kernel, and memory used by important applications, but use a much lower level of protection for processes of less importance.

Throughout their paper Yoon and Erez present various optimizations, for example an optimized use of the processor cache for ECC checksums. Additionally,

they evaluated the performance impact of their implementation, and its influence on power consumption. Summarized, their design had a negligible impact on performance, with runaway values topping at a normalised runtime of 1.7 for the GUPS streaming benchmark.
Remarkably, the Energy Delay Product (EDP) of a system with virtualized ChipKill protection is often lower, than the EDP for systems with native ChipKill. They explained this with the fact, that the DDR-modules with $x4$ chips used in ChipKill systems are less power efficient, than modules with $x8$ or $x16$ chips.

3.3.2.1 Relation to this Thesis

Yoon's and Erez' research is an important step on the road towards more reliable computer systems. It remains to be seen, if, and when the first real world implementations of their design will be seen. Regarding this diploma thesis, the usage of sophisticated *in situ* memory error detection (and correction) decreases the chance that (undetected) errors corrupt data. Software based memory tests increase the chance, that hardware errors that exceed the capability of the underlying systems, are detected.

Chapter 4

Solution: A Memory Testing Architecture for Linux

An important step in the design and implementation of software is the gathering of requirements. Requirements describe the properties the final product should, or must have. A minimal requirement for an online memory tester surely is "The software should prevent more errors than it causes" – but this requirement is too unspecific, and leaves out many important aspects.

In this chapter the requirements for an online memory tester are gathered. The collected requirements are the base of the implementation design, that is then presented in section 4.2.

4.1 Requirements

The following sections investigate three functional aspects: fault detection for different fault models, the cost-benefit ratio, and the desire to provide the foundation for further research efforts. Further, three software engineering related aspects are investigated. These technical aspects have their roots in the functional requirements, as will be discussed.

4.1.1 Detection of Errors

The most important requirement of the memory tester is, that it should be able to detect functional faults and defects in memory. It is not required though, that *every* fault (defect) must be detected, e.g. faults in memory that cannot be accessed by the implementation, faults not covered by the selected fault model, or faults that happen in memory locations that are currently not tested cannot be required to be detected.

Requirement 1: Detection of Errors *The implementation must be able to detect memory errors.*

Requirement 2: Isolation of Errors *The implementation has to ensure, that memory which has been detected as faulty must not be used any more.*

From the analysis of the Linux mm in section 2.4 it can be concluded, that it is unrealistic that the implementation can test all of the physical memory. The implementation should however, be able to test a substantial amount of the installed physical memory.

Requirement 3: Memory Coverage *The implementation should be able to test a substantial amount of the physical memory.*

This requirement can further be broken down into sub-requirements that specify this into more detail.

Requirement 4: No Test of Kernel Memory *The implementation does not need to be able to test the physical memory that is used by the kernel for its internal data and code.*

Requirement 5: Testing Free Buddy Memory *The implementation should be able be able to test memory, that is marked as free by the buddy allocator (see section 2.4.2.9).*

Claiming memory that the kernel uses for its internal processing is very difficult, and is probably not practical to do. Further, the Linux kernel only uses the first GiB of physical memory for its internal structures, so there is an upper bound for the memory not accessible because of this requirement [62, 46, 53]. Requirements 6 and 7 explicitly exclude memory that is connected to modified data, because it is not always possible to reliably interfere with the Linux kernel, when it handles memory in this state [79].

Requirement 6: Testing Clean Anonymous Memory *The implementation should be able be able to test memory that is used as anonymous memory. It is not necessary, that memory that is backed by swap space or memory mapped files can be tested.*

Requirement 7: Testing Clean Page-Cache Memory *The implementation should be able be able to test memory, that is used in the Linux page cache. Memory that is marked as dirty, or memory that is currently used by I/O does not need to be tested.*

An exact quantification of the word *substantial* in requirement 3 cannot be given, because this depends on the total amount of memory installed, the work load of the system, and the dynamic behaviour of the Linux mm. Visualising the dynamics of the mm, especially the usage patterns of each frame can help to understand, how the mm behaves (see requirement 15), and if additional or different strategies for frame test scheduling are necessary.

4.1.1.1 Different Fault Models

Different fault models for RAM hase been discussed in section 2.2. The fault models presented all have their respective advantages and disadvantages. The most notable variable of the presented fault models is the complexity of the test that detects the faults described by the model. Re-implementing the memory test algorithms presented by various researchers is not within the scope of this diploma thesis. On the other hand, an evaluation of the feasibility of a memory test requires the existence of a memory test. This leads to the following requirements:

Requirement 8: Multiple Fault Models *The implementation must support multiple fault models. It is sufficient, if only a restricted subset of all possible fault models can be supported by the implementation.*

Requirement 9: Emulating of Fault Models *The implementation must implement an emulation that resembles the runtime complexity of two fault models.*

Except for very reduced fault models, most fault models incorporate faults that affect multiple memory cells, and faults where memory cells influence each other [118, 102]. The concept of NPSFs (Neighbourhood Pattern Sensitive Faults, see section 2.2) should, with restrictions regarding the structure of the neighbourhoods, be supported by the implementation. Restricting the structure of the neighbourhoods is owed to the observations made in section 2.4, and the restrictions allowed by the requirements 4 through 7.

Requirement 10: Support Neighbourhoods *The implementation must support fault models that test for NPSFs. The implementation is allowed to restrict size and composition of the neighbourhoods.*

4.1.2 Cost-Benefit Ratio

A major goal of this diploma thesis is to determine, whether online memory tests are feasible or not. The difficulty of defining an objective meaning of *feasible* have been discussed in section 1.4. The implementation of the memory tester should *enable* the researcher to answer the question of feasibility by allowing the researcher to collect runtime metrics. This leads to the following requirements:

Requirement 11: Cost-Benefit Estimation *By benchmarking the application it should be possible to estimate the cost of an online memory test.*

Reserving memory for a memory test could be a major bottleneck, considering the fact, that this is a deep intervention in the inner workings of the mm, and could seriously affect performance by bringing the fine tuned system of lock, cache and TLB handling out of balance.

Requirement 12: Cost-Benefit Estimation: Cost of Acquisition *The implementation should provide performance metrics that indicate how expensive the acquisition of certain memory areas has been.*

Another potential bottleneck is the actual test of a memory area, the impact of this should be measurable as well:

Requirement 13: Cost-Benefit Estimation: The Cost of Tests *By benchmarking the application it should be possible to estimate, how big the impact of memory testing is.*

4.1.3 Base for Further Research

It is desirable, that this diploma thesis, and the accompanying implementation, serves as the basis for further research. For this, it is preferable to allow as many aspects of the implementation as possible to be exchangeable.

Requirement 14: Enabling Further Research *The implementation should be written in such a way, that further further research can adopt and change it easily.*

The requirement stated in number 14 is a very fuzzy requirement that is not easy to measure. In the next section several technical, non-functional requirements are discussed. These requirements support the functional requirements 1 through 14, and help to assess the implementation choices with regard to the requirements.

4.1.3.1 Visualisation

To understand, what a system does, and how its behaviour changes over time, it is often helpful to visualize dynamic processes and state changes. A meaningfull visualization can inspire, and support additional research.

Requirement 15: Visualization of Memory *The implementation should provide the means to visualize the state of memory over time.*

4.1.4 Performance

Requirements 11 – 13 emphasize the aspect of *performance*. As already discussed in section 1.4, the word performance is not very clearly defined, and a concrete definition depends on the context it is made in. In the case of a memory tester, performance can mean anything from *number of bytes tested per second*, or *accession of the EDP*, up to *influence on the Transactions Per Seconds metric of the database.*

Performance is an important issue, but in the context of this thesis, the focus of attention is not on performance. As Donald Knuth said famously “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil”[80]. Experience showed that this statement still holds, and that targeted high level optimizations are far more likely to improve the overall performance of a system.

Requirement 16: Enabling Bottleneck Detection *The implementation should be written in such a way, that it is feasible to detect performance bottlenecks.*

Requirement 17: Enabling High-Level Optimizations *The implementation should be written in such a way, that it is feasible to apply high level performance optimizations.*

4.1.5 Extensibility

In requirement 8 the possibility to implement multiple fault models is requested. Requirement 3 requests, that as much as possible of the system memory should be testable. Both requirements can be supported by designing the implementation in such a way, that it is extensible. Adding new methods to acquire memory from the kernel can increase the test coverage, designing the fault models as strategy patterns [61] allows the implementation of multiple fault models.

Requirement 18: Extensible System *The implementation should be written in such a way, that it easily extensible.*

This requirement (18) can be realised by designing a modular architecture:

Requirement 19: Modular Design *The implementation should be designed in a modular fashion.*

4.1.6 Stability

Section 1.1 extensively discussed the problems that can be caused by malfunctioning software. Software that closely interacts with, and even modifies internal structures of the kernel, must be very reliable. Errors in a memory tester implementation can lead to serious data corruption – the very same effect, that it should prevent.

By isolating the parts of the implementation from each other, the stability of the system can be enhanced:

Requirement 20: Fault Isolation *The components of the system should be isolated from each other, so that faults remain local to single components.*

Requirement 21: Memory Leaks *The design must ensure that memory leaks are effectively prevented.*

Requirement 22: Principle of Least Privilege *No component should have more privileges, than it absolutely needs to function.*

By allowing the implementation to be automatically tested, the reliability of the implementation can be increased:

Requirement 23: Testability *It should be easy to test the implementation.*

Requirement 24: Maintainability *The implementation should be easily maintainable, and be impervious to changes in the kernel.*

Closely related to the issues of stability is the question of breaking the mm's encapsulation of its data structures. If at all possible, internal structures of the kernel should not be interpreted or modified. The rapid change in the Linux-mm – see section 2.4.3 – makes it very difficult to keep a stable implementation.

Requirement 25: Kernel reuse *The implementation should not break the encapsulation of the kernel's internal data structures, unless absolutely neccessary.*

4.2 Design

The design elaborated for the implementation of the accompanying online memory tester aims to satisfy the requirements presented in section 4.1. This section describes the components that form the implementation, how they relate to each other, and why this design has been chosen.

4.2.1 Guiding Ideas

The guiding ideas of the design are to maximize the amount of code that runs in user space, and to minimize the amount of code running in kernel space. Secondly, as much of the implementation should be designed in such a way, that it is easily replaceable.

Bugs in kernel code can lead to fatal crashes and data corruptions, this is why kernel code should be reduced as much as possible. User space programs allow a much finer adherence to the principle of least privilege (requirement 22), and the isolation of processes allows a better fault isolation (requirement 20).

Developing user space programs is considered to be significantly easier than writing and debugging kernel level code. This lowers the gateway hurdle for modifications and extensions (requirements 8+9, 14, 15, 18+19), and greatly eases testing (23).

Running code in user space also eases the task of profiling the implementations performance (requirements 11–13, and 16+17), because profiling of user space components is often easier than profiling kernel components.

4.2.2 Overview of the Design

Figure 4.1 presents the components of the implementation. A major focus had been laid on a modularized design. Core components like test algorithms for fault models (compare requirements 1 and 8), and the difficult task of acquiring specific memory regions from the kernel (requirements 3–7) are implemented as strategy patterns [61].

All code running in the kernel is designed as a kernel module. This decision supports encapsulation and modularity (requirement 19), and increases main-

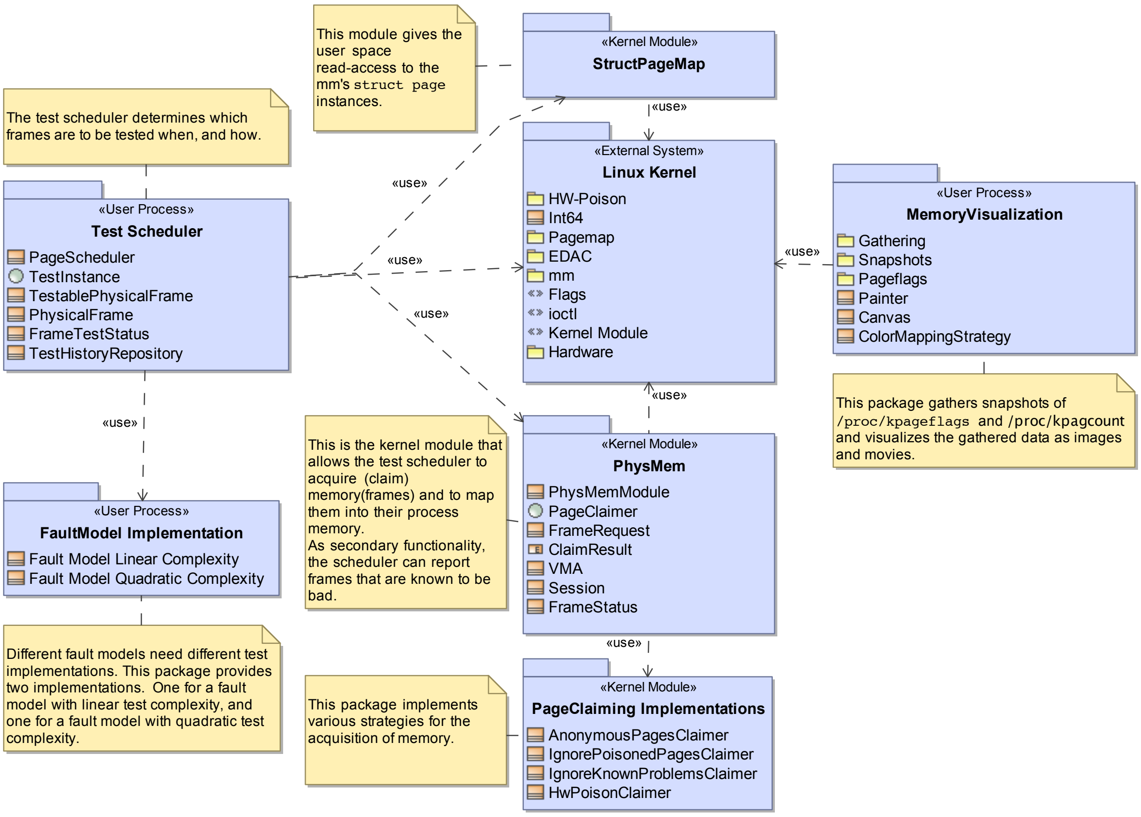

This module gives the user space read-access to the mm's struct page instances.
«Kernel Module»
StructPageMap
The test scheduler determines which frames are to be tested when, and how.
«use»
«External System»
Linux Kernel
HW-Poison
Int64
Pagemap
EDAC
mm
Flags
ioctl
Kernel Module
Hardware
«User Process»
Test Scheduler
PageScheduler
TestInstance
TestablePhysicalFrame
PhysicalFrame
FrameTestStatus
TestHistoryRepository
«use»
«use»
«use»
«use»
«User Process»
MemoryVisualization
Gathering
Snapshots
Pageflags
Painter
Canvas
ColorMappingStrategy
This package gathers snapshots of /proc/kpageflags and /proc/kpagcount and visualizes the gathered data as images and movies.
«use»
«use»
«User Process»
FaultModel Implementation
Fault Model Linear Complexity
Fault Model Quadratic Complexity
This is the kernel module that allows the test scheduler to acquire (claim) memory(frames) and to map them into their process memory.
As secondary functionality, the scheduler can report frames that are known to be bad.
«Kernel Module»
PhysMem
PhysMemModule
PageClaimer
FrameRequest
ClaimResult
VMA
Session
FrameStatus
«use»
Different fault models need different test implementations. This package provides two implementations. One for a fault model with linear test complexity, and one for a fault model with quadratic test complexity.
This package implements various strategies for the acquisition of memory.
«Kernel Module»
PageClaiming Implementations
AnonymousPagesClaimer
IgnorePoisonedPagesClaimer
IgnoreKnownProblemsClaimer
HwPoisonClaimer

Figure 4.1: *Package Diagram of the Design*
The package diagram of the implementation shows seven packages, six of them are a part of the implementation.
The `TestScheduler`, on the left side, determines which frames are to be tested when, and how. Algorithms for different fault models are implemented according to the strategy pattern.
Kernel based services are located in the middle column. At the top is the `StructPageMap` package that allows user space processes to *mmap* the page-flags into their address room (see section 4.2.2.1). Below it are the Linux kernel and the `PhysMem` module which implements the functionality, giving the user space access to the frames acquired by the `PageClaiming Implementations`.
On the right side, the `MemoryVisualization` package can collect snapshots of the page-flags and generate videos that visualize the behaviour of the mm.

tainability (requirement 24), but also has drawbacks. The kernel subsystems, especially the mm, protect their internal data structures, and the routines to modify them by not exporting them to kernel modules [53, 87]. Routines that manipulate these structures need to be able to access them, and this requires the kernel to be patched to export these symbols (compare to section 5.3.1).

Memory is managed in frame granularity, the same granularity the Linux mm uses to manage memory. On $x86$ systems the size of frames is $4KiB$, the larger frame sizes of $2MiB$, and $4MiB$ are not supported by the implementation. This fulfills the requirement for neighbourhoods, as stated in requirement 10.

4.2.2.1 Reusing Kernel Functionalities

Given the complexity and the volatility of the kernel's behaviour and implementation, it is important to reuse as much as possible of the kernel's existing infrastructure. Two central needs of the implementation are good examples for kernel-reuse.

The decision which frame should be tested when is made by the `PageScheduler` (see figures 4.1 and 4.3). To determine the optimal set of frames to be scheduled for testing, the scheduler must balance two factors: the cost of acquiring the frame, and the criticality that the frame is tested by the next test. The later value mainly depends on how long this frame has not been tested. The former, the cost estimation, heavily depends on the current usage of the frame. With the experience of how cost intensive the acquirement of similar frames had been in the past, the future cost of acquisition can be estimated.

An important indicator of the to be expected cost are the page-flags the kernel uses to manage frames [62]. The *pagemap* patches [11] allow the user space to access the page-flags by reading the `/proc/kpageflags` file. Although this access is rather slow because each read access results in a systemcall, it is preferred over the `StructPageMap` module that allows the page-flags to be *mmap*ed into the caller's address space. `StructPageMap` has been implemented as a part of this diploma thesis, but because of the re-use requirement (25) its usage has been replaced by `/proc/kpageflags`.

Requirement 2 states that frames that have been found to be faulty must not be used again by the kernel. By leveraging the HW-Poison patches [79, 60], the implementation can fulfill this requirement with relative ease.

4.2.3 Design Decisions

The act of writing software often includes deliberate decisions for, or against tools, programming languages, and design principles. Several decisions have been made in the planning process of the implementation. Two of them, the decision to use the scripting language Python [19] for the user space implementation, and the decision how the user space implementation interacts with the kernel space implementation are worth a further discussion.

4.2.3.1 Implementing the User Space in Python

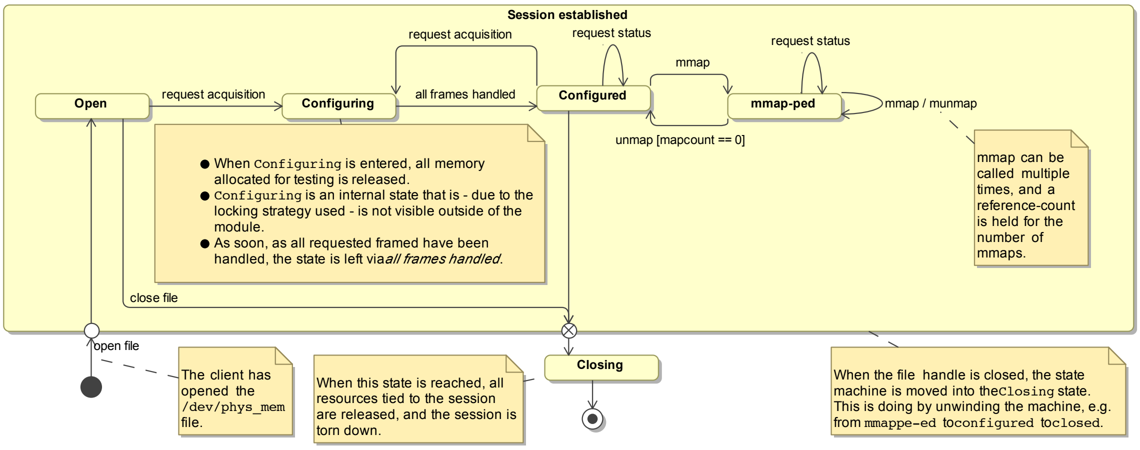
Python is an object oriented scripting language available on a variety of operating systems. It is easy to learn, allows rapid development, test driven design, and can use the system calls used for the kernel implementation [18, 17]. Further it allows to incorporate C libraries [16] and is supported by the C interface compiler SWIG [21].

Implementing the user space, especially the test routines of the fault models, in a scripting language seems counterintuitive, if the goal of the implementation is to evaluate the performance impact of a memory tester. In section 4.1.4 it has already been discussed, that the performance of the implementation is of secondary importance. The decision for python had been made because it seemed that the benefits of using a high level scripting language by far outweigh the performance penalty.

4.2.3.2 Tying the Acquired Memory to the Caller Process

The kernel implementation that acquires memory for the caller ties the acquired memory to the calling process. This has the advantage, that a process that acquires memory will free the memory latest when it terminates, or earlier. Although this makes an interaction between a scheduler process and a memory tester implementation running in a different process more difficult, it prevents memory leaks (requirement 21).

Implementation wise, this is done by associating a session to the file descriptor the client uses to communicate with the kernel module. Memory allocated to a session is freed as soon, as the file handle closes. Figure 4.2 shows the



- When `Configuring` is entered, all memory allocated for testing is released.
- `Configuring` is an internal state that is - due to the locking strategy used - is not visible outside of the module.
- As soon, as all requested framed have been handled, the state is left via*all frames handled*.

mmap can be called multiple times, and a reference-count is held for the number of mmaps.

The client has opened the `/dev/phys_mem` file.

When this state is reached, all resources tied to the session are released, and the session is torn down.

When the file handle is closed, the state machine is moved into the`Closing` state. This is doing by unwinding the machine, e.g. from `mmappe-ed` to`configured` to`closed`.

Figure 4.2: *State Diagram of the Session*
This state chart shows, how the users session is handled inside the `PhysMem` module. The state of the session determines the reaction to requests such as *mmap*/*munmap*, *configure* or *close*.
By opening the `/dev/phys_mem` device, the client implicitly opens the session. The next step of the client is to send a *request for frames*. This causes the session to enter the `Configuring` state. The frame acquisition is then handled inside the `Configuring` state. Once all frame requests have been handled, the session moves to the `Configured` state, where the client can use the file descriptor to request the result of its request, and to *mmap* the acquired memory into its address space.

states, the session can be in, and how the proper release of allocated memory is enforced by unwinding the session state-machine of the, if the file handle is closed.

4.2.4 Scheduler & Frame Acquisition

Figure 4.3 focuses on the test-, or frame-scheduling component. To fulfill the requirements of flexibility, and extensibility, the scheduler is designed to make heavy use of decoupling patterns such as *Dependency Injection*, and *Strategy Objects*. Although not stated by the requirements, the scheduler persists its internal state and the knowledge it gathered about individual frames, and frame acquisition costs. This allows the scheduler to be shut down, and restarted

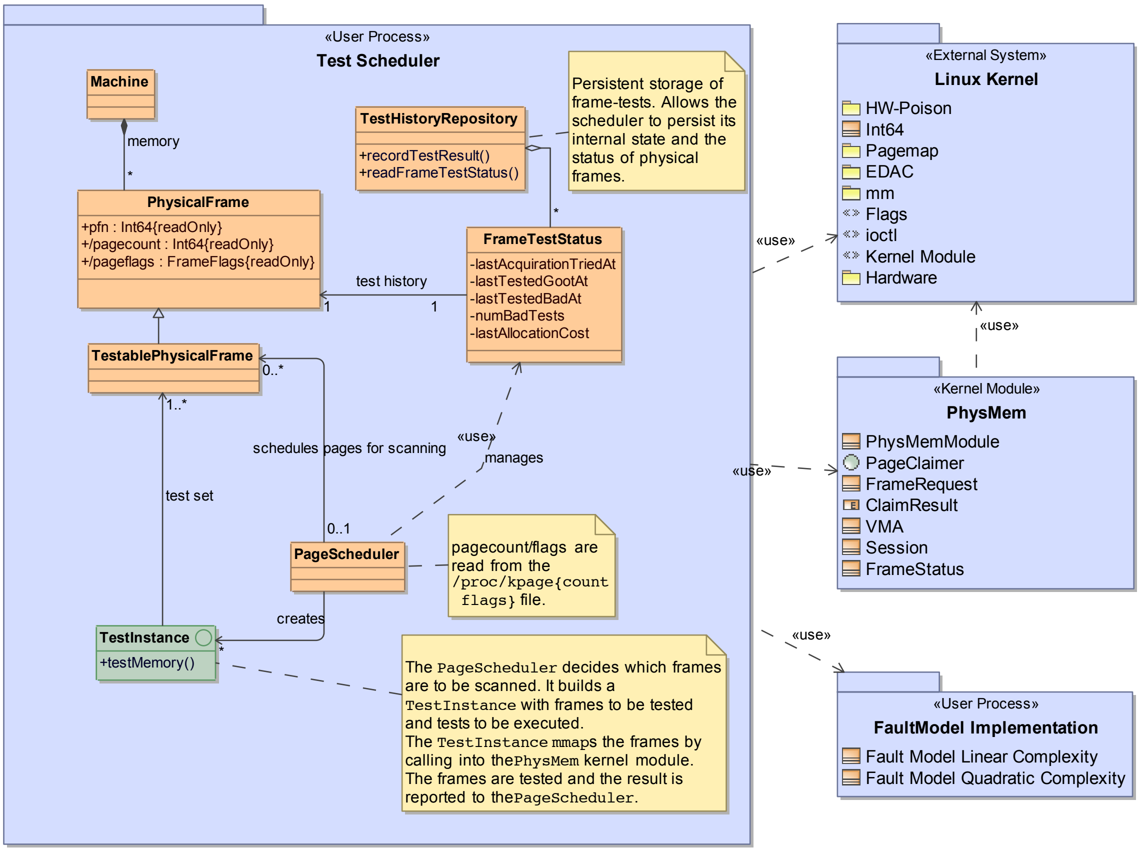
```
 1 struct phys_mem_frame_status {
 2    struct phys_mem_frame_request request;
 3    /* A pointer to the first byte of the frame, relative to the start of the VMA */
 4    unsigned long long            vma_offset_of_first_byte;
 5    // ...
 6    /* How long did it take to get a hold on this frame? Measured in jiffies*/
 7    unsigned long long            allocation_cost_jiffies;
 8    /* A single item of SOURCE_* (optionally ORed with one SOURCE_ERROR_*) */
 9    unsigned  long                actual_source;
10 };
11
12 struct phys_mem_frame_request {
13    unsigned   long requested_pfn;
14    unsigned  long allowed_sources; /* Bitmask of SOURCE_* */
15 };
16   // ....
17
18   struct phys_mem_frame_request frame_requests[20];
19   // ... fill the frame_requests and choose for each individial frame which claiming strategy should be
         used, by assigning a bitmap of sources to the allowed_sources fields.
20   struct phys_mem_request r;
21   r.num_requests=20;
22   r.protocol_version=1;
23   r.req = &frame_requests;
24
25   ret = ioctl(dev, IOCTL_REQUEST,&r);
26   // ...
27   // the file descriptor now allows the client
28   // to read and seek in the file.
29   // The file content is an array of  r.num_requests
30   // struct phys_mem_frame_status instances
31
32   void* mem = mmap(0, num_successfully*PAGESIZE, ...,dev,0);
33   // mem now points to the acquired memory
```

Listing 4.1: *Requesting Memory From the Module*
This listing exemplifies, how a client could acquire 20 specific frames from the kernel module. Lines 1–16 contain some of the data types used. The actual request is conducted in line 25. Not shown is the part where the client reads the file *dev* and collects the result of the request by reading the *struct phys_mem_frame_status* instances from the file.

without losing important historic data.

To give the scheduler more control over the acquisition process, it can control, which strategies are used to acquire each frame. This allows the scheduler to adapt to the circumstances, e.g. the scheduler can choose a slower but more capable claiming strategy, if it finds this is needful. See listing 4.1 for an example. In addition to choose the acquisition strategy, the scheduler can also choose, and change, the fault model at runtime, and individually for each frame.

Figure 4.4 focuses on the kernel side of the design. The key concepts here are strategies for frame acquisition, strong coupling of clients to a session, and using *mmap* to allow the client to access the memory.



Persistent storage of frame-tests. Allows the scheduler to persist its internal state and the status of physical frames.

pagecount/flags are read from the `/proc/kpage{count flags}` file.

The `PageScheduler` decides which frames are to be scanned. It builds a `TestInstance` with frames to be tested and tests to be executed.
The `TestInstance` mmaps the frames by calling into the`PhysMem` kernel module.
The frames are tested and the result is reported to the`PageScheduler`.

Figure 4.3: *Package Diagram of the Design, focused on the Test Scheduler*
This diagram focuses on the `Test Scheduler` component, and is a focused view on figure 4.1.
The `Test Scheduler` component can be broken into core basic parts: A view on the current state of frames (`PhysicalFrame`), persistent historic meta-data about each memory frame (`TestHistoryRepository, FrameTestStatus`), test routines that implement the fault models (Interface `TestInstance`, implemented by strategies in package `FaultModel Implementation`), and the central scheduling implemented in the `PageScheduler`.
The page scheduler determines the frames to be tested using the current status of the frame, and historic information about the frame, e.g. time since last successful test, number of errors found in the frame, etc.
By persisting the meta-data in `FrameTestStatus`, the scheduler can be shut down and restarted without losing this data.

4.2.5 Memory Visualization

The objective of the memory visualization is to create images or videos that visualize the usage of system memory over time. This diploma thesis implements two visualizations: One implementation to track the mapcount of a page, that is, how often the frame is mapped into memory. The other implementation interprets the pageflags, and maps them into an image. By altering the interpretation of the frame flags, different aspects of the mm can be highlighted. E.g., by specially marking dirty (not written back to storage) pages, the distribution of these pages can be tracked. By highlighting all frames used by anonymous memory, the user can track the memory usage of processes. Figure 4.5 is an example for such an interpretation.

In order to decouple data gathering and image processing, both tasks are split up into two different programs. A shell script is used to gather data, and a python application batch processes the data into images, which then can be stitched together to a video.

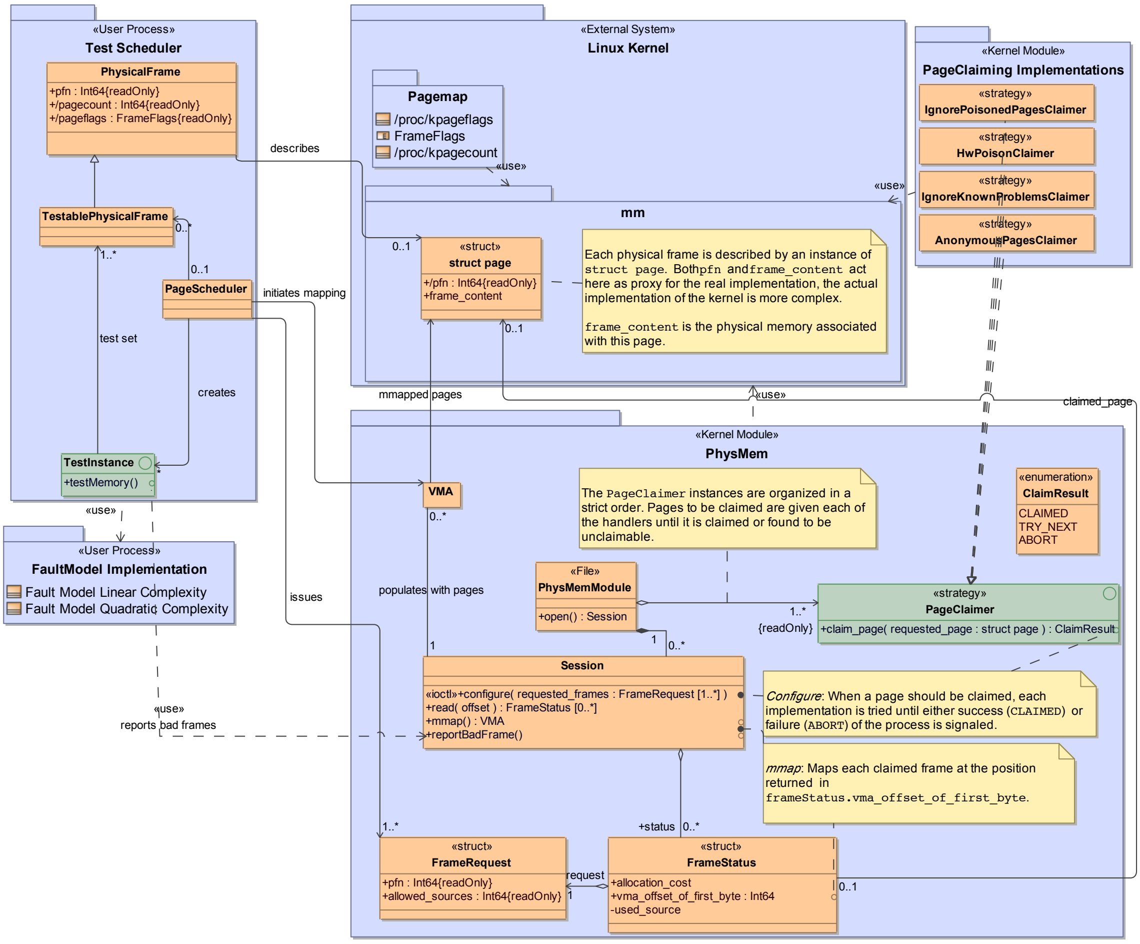

«User Process»
Test Scheduler
PhysicalFrame
+pfn : Int64{readOnly}
+/pagecount : Int64{readOnly}
+/pageflags : FrameFlags{readOnly}
TestablePhysicalFrame
0..*
1..*
0..1
PageScheduler
test set
creates
TestInstance
+testMemory()
«use»
«User Process»
FaultModel Implementation
Fault Model Linear Complexity
Fault Model Quadratic Complexity
«use»
reports bad frames
describes
initiates mapping
issues
«External System»
Linux Kernel
Pagemap
/proc/kpageflags
FrameFlags
/proc/kpagecount
«use»
«use»
mm
0..1
«struct»
struct page
+/pfn : Int64{readOnly}
+frame_content
0..1
Each physical frame is described by an instance of struct page. Both pfn and frame_content act here as proxy for the real implementation, the actual implementation of the kernel is more complex.
frame_content is the physical memory associated with this page.
mmapped pages
«use»
«Kernel Module»
PageClaiming Implementations
«strategy»
IgnorePoisonedPagesClaimer
«strategy»
HwPoisonClaimer
«strategy»
IgnoreKnownProblemsClaimer
«strategy»
AnonymousPagesClaimer
claimed_page
«Kernel Module»
PhysMem
VMA
0..*
The PageClaimer instances are organized in a strict order. Pages to be claimed are given each of the handlers until it is claimed or found to be unclaimable.
«enumeration»
ClaimResult
CLAIMED
TRY_NEXT
ABORT
populates with pages
«File»
PhysMemModule
+open() : Session
1..*
{readOnly}
«strategy»
PageClaimer
+claim_page(requested_page : struct page) : ClaimResult
1
1
0..*
Session
«ioctl»+configure(requested_frames : FrameRequest [1..*])
+read(offset) : FrameStatus [0..*]
+mmap() : VMA
+reportBadFrame()
Configure: When a page should be claimed, each implementation is tried until either success (CLAIMED) or failure (ABORT) of the process is signaled.
mmap: Maps each claimed frame at the position returned in frameStatus.vma_offset_of_first_byte.
1..*
+status
0..*
«struct»
FrameRequest
+pfn : Int64{readOnly}
+allowed_sources : Int64{readOnly}
request
1
«struct»
FrameStatus
+allocation_cost
+vma_offset_of_first_byte : Int64
-used_source
0..1

Figure 4.4: *Package Diagram of the Design, Focused on the PhysMem Module*
Despite the seemingly complex class diagram, the PhysMem kernel module is designed in a very straight forward manner. The client can request frames, and map the acquired frames into their address space. In this context, the strategies for frame acquisition are called *sources*. After the request has been processed, the client reads the result of the request via the *read* call of the file handle. The returned data contains metrics about the request. The client accesses the memory by *mmap*. If a client detects a faulty frame, it uses the reportBadFrame method of the Session, the memory frame is then offlined by utilizing the HW_POISON module.
Not shown is the design of the mapping from calls to the file interface to the Session interface. This is designed to be implemented by state-specific adapters from the file interface to the Session interface.

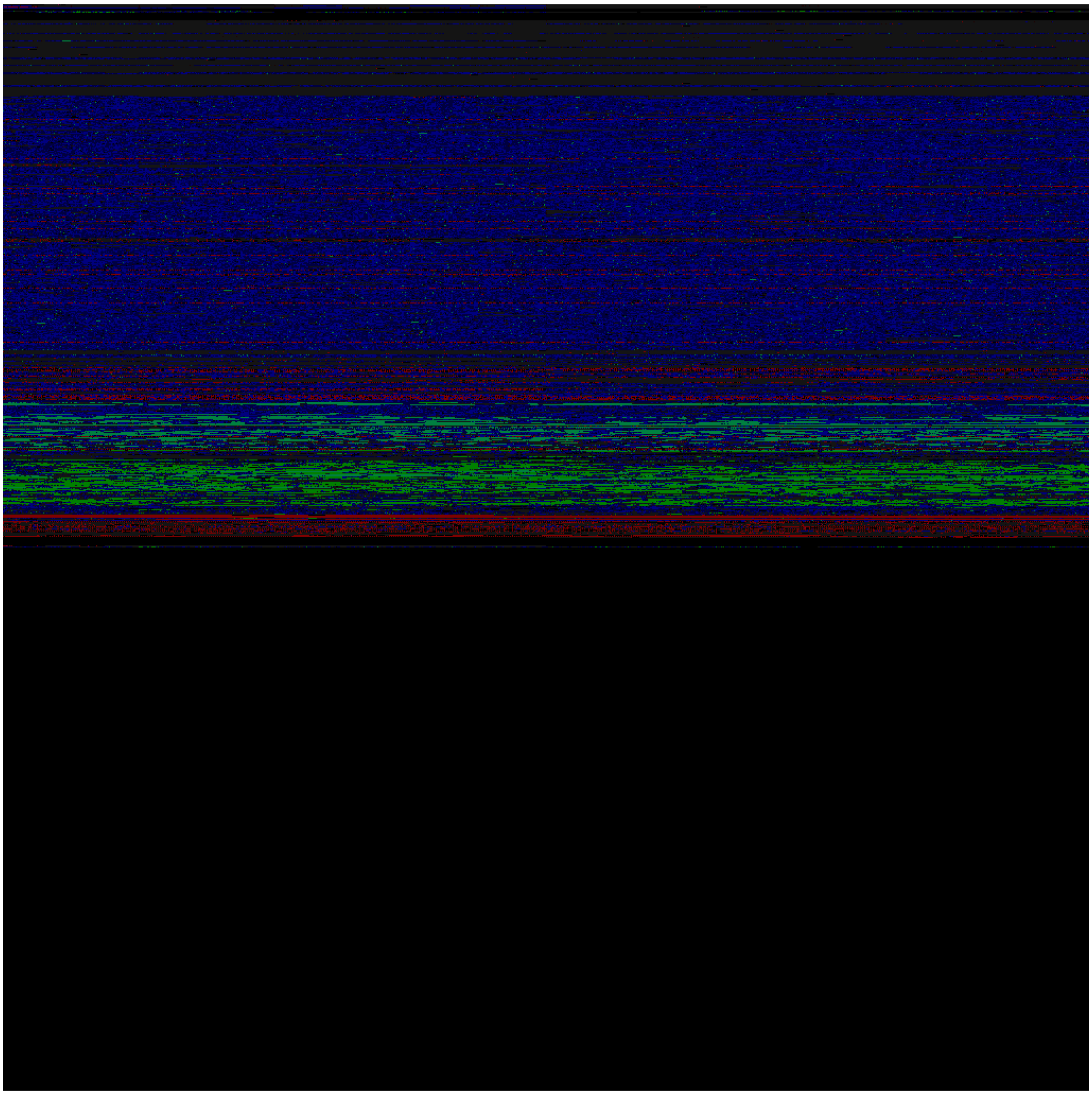


Figure 4.5: *Example of Visualised Memory Usage*
This image is a snapshot of a machine with $2GiB$ of RAM while it compiles the Linux kernel. Each pixel resembles one frame. Green pixels are anonymous memory, blue pixels are active frames, and red frames are free frames. Black pixels are either not available (the bottom half), or are not textured by the used visualization algorithm.

Chapter 5

Implementation

Chapter 5 discusses, how the implementation reflects the requirements, and the design stated in chapter 4. Further areas, that posed difficulties – or opportunities – are discussed. The first part discusses implementation details of the user- and kernel-space. The second part discussed challenges and a case, where a part of the Linux kernel that was meant to be central to the implementation turned out to be unusable.

5.1 Kernel Space

Section 4.2 stated, that the implementation inside the kernel should be as simple, and as safe as possible. This section highlights two parts of the implementation. The first shows, how the state machine described in figure 4.2 is implemented. The second part reviews the implementation that handles the actual acquisition of frames. The naming convention in the source code uses a slightly different naming, for this discussion the terms *acquisition* and *claim* are used synonymously.

5.1.1 Handling the Session State Machine

The contract of the kernel module states, that a client has to request a batch of frames for testing, and that the client can *mmap* these frames into its address space only after the session is configured.

```
int phys_mem_open (struct inode *inode, struct file *filp)
{
  // ...
  sema_init(&session->sem,1);
  // ...
  session->status.state = SESSION_STATE_INVALID;
  SET_STATE(session, SESSION_STATE_OPEN);
  /* and use filp->private_data to point to the session data */
  filp->private_data = session;
  // ...
}

int phys_mem_release (struct inode *inode, struct file *filp)
{
  // ...
  session = (struct phys_mem_session*) filp->private_data;
  // Depending on the state of the session we need to do several
  // steps. This "fast'n'furious" implementation of the state
  // machine is chosen over the 'real' steps because
  // this way it is much cleaner.
  while (  GET_STATE(session) != SESSION_STATE_CLOSED) {
    switch(GET_STATE(session)) {
      // ...
    case SESSION_STATE_OPEN:
      SET_STATE(session, SESSION_STATE_CLOSED);
      break;

    case SESSION_STATE_CONFIGURING:
       // ...
    case SESSION_STATE_CONFIGURED:
       // Free all claimed pages
      free_page_stati(session);
      SET_STATE(session, SESSION_STATE_OPEN);
      break;

    case SESSION_STATE_MAPPED: break;
      // ...
      SET_STATE(session, SESSION_STATE_CONFIGURED);
      break;
    }
  }
  // ...
}

//
/**
 * The file operations indexed by the session state. Dispatcher functions
 * extract the sessionstate from the file* and call the matching implementation
 * fops_by_session_state[status].op
 *
 */
struct file_operations fops_by_session_state[] = {
 {
  /* CLOSED */
 .llseek = NULLi, .read = NULL, .ioctl = NULL, .mmap = NULL,
 },
 {
 /* OPEN */
 .llseek = NULL, .read = NULL, .ioctl = file_ioctl_open, .mmap = NULL,
 },
 {
 /* CONFIGURING */
 .llseek = NULL, .read = NULL, .ioctl = NULL, .mmap = NULL,
 },
 {
 /* CONFIGURED */
 .llseek = file_llseek_configured, .read = file_read_configured, .ioctl = file_ioctl_open, .mmap =
       file_mmap_configured,
 },
 {
 /* MAPPED */
 .llseek = file_llseek_configured, .read = file_read_configured, .ioctl = NULL, .mmap =
       file_mmap_configured,
  },
};
```

Listing 5.1: *Session State Handling*
This listing shows, how the file-operations are handled by the state machine, and how the proper termination of the session is enforced in *phys_mem_release*.

To allow a clean structure and assertable preconditions in the handler routines, an automaton has been implemented (listing 5.1). Systemcalls on the file-handle are dispatched via tables of virtual methods, similar to how *vtables* are used in many C++ compilers to implement virtual methods. The most interesting method is *phys_mem_release*, as it unwinds the state machine from any state to the closed state.

5.1.2 Handling the Frame Acquisition

Listing 5.2 shows the method that moves the statemachine from `Open` through `Configuring` to `Configured`. The method is implemented in a loop that iterates over each frame requested by the client to acquire (claim) the frame. The actual frame acquisition is implemented by the strategy objects. See line 6 for the strategy objects configured for the depicted build.

Each strategy implementation verifies the `allowed_sources` bitmask passed by the client. A strategy those id is not included in the allowed sources simply returns `CLAIMED_TRY_NEXT`.

The whole implementation is very simple and modular, allowing the developer to extend and modify the implementation with relative ease.

5.1.3 Reflection on the Kernel Implementations Design

It turned out, that the design allowed the kernel module, and the frame acquisition strategies, to be tested from the user space. This greatly helped the development and is a contribution to requirement 23.

Further, the design lends itself to an explorative implementation of a kernel based implementation of a fault model. This implementation has not been incorporated into the final version because its performance did not differ from a likewise explorative user space implementation of the same fault model. Having seen, that a fault-detection routine running in user space is not significantly slower than a routine running in kernel mode backs up the decision to locate the actual test in user space.

```
1  /**
2   * These are the methods that the module uses to claim a page.
3   */
4  // try_claim_method try_claim_methods[]  =  {try_claim_free_page,try_claim_free_buddy_page,
         try_claim_page_in_page_cache,try_claim_page_from_user_process, NULL};
5  //  try_claim_method try_claim_methods[]  =  {try_claim_free_buddy_page,NULL};
6  try_claim_method try_claim_methods[]  =  {try_claim_free_buddy_page, try_claim_page_via_hwpoison, NULL};
7
8  int handle_request_pages(struct phys_mem_session* session, const struct phys_mem_request* request)
9  {
10 // ...
11     {
12       /* The VMA maps all successfully mapped pages in the same order as they appear here.
13        * To make the users live easier, the relative offset of the frames gets returned in
               vma_offset_of_first_byte.
14        */
15       // ...
16       for (i = 0; i < request->num_requests; i++) {
17         struct phys_mem_frame_request __user  * current_pfn_request = &request->req[i];
18         struct phys_mem_frame_status __kernel * current_pfn_status = &session->frame_stati[i];
19
20         if (copy_from_user(&current_pfn_status->request, current_pfn_request, sizeof(struct
               phys_mem_frame_request))){
21             // .. FAIL
22         }
23
24         /* current_pfn_status now points to a blank page status
25          */
26          jiffies_start = get_jiffies_64();
27
28         if (unlikely(!pfn_valid(current_pfn_status->request.requested_pfn))) {
29            // .. FAIL
30         }else{
31           struct page* requested_page = pfn_to_page(current_pfn_status->request.requested_pfn);
32
33           if (unlikely(NULL == requested_page)){
34           // .. FAIL
35           }
36           try_claim_method claim_method = NULL;
37           // ...
38           /* Iterate each method until a) success or b) failure or c) no more methods */
39           while (CLAIMED_TRY_NEXT == claim_method_result) {
40             claim_method = try_claim_methods[claim_method_idx];
41
42             if (claim_method)
43               claim_method_result = claim_method( requested_page, current_pfn_status->request.
                     allowed_sources, &allocated_page, &current_pfn_status->actual_source);
44             else
45               claim_method_result = CLAIMED_ABORT;
46
47             claim_method_idx++;
48           }
49           if (CLAIMED_SUCCESSFULLY == claim_method_result){
50             current_pfn_status->pfn = page_to_pfn(allocated_page);
51             current_pfn_status->page = allocated_page;
52             current_pfn_status->vma_offset_of_first_byte = current_offset_in_vma;
53             current_offset_in_vma += PAGE_SIZE;
54           } else {
55             /* Nothing to do, reset object*/
56             // ...
57           }
58         }
59         // Record the time it took to acquire this frame
60         current_pfn_status->allocation_cost_jiffies = jiffies_used;
61       }
62     }
63     SET_STATE(session, SESSION_STATE_CONFIGURED);
64     // ...
```

Listing 5.2: *Handling Frame Acquisition*

This listing shows, how a clients request is handled. The loop from line 16–61 iterates over each requested frame and tries to acquire the frame in line 43.

```
 1 $ # start the memory test
 2 $ ./main.py
 3
 4 0x230000 frames (2293760 decimal), of which are 945815 tested  (41.2 %) and 1347945 untested  (58.8 %),
         0 have seen errors.
 5 For tested frames, the following statistics have been calculated:
 6   Time it took to claim a frame (in jiffies) (min,max,avg) : 0, 1, 0
 7   Timestamp of last test (min,max,avg) : 2010-06-03 22:30:22, 2010-06-04 11:23:32, 2010-06-04 05:03:00
 8 ...
 9 ^C
10 KeyboardInterrupt
11 $ ./main.py
12
13 0x230000 frames (2293760 decimal), of which are 945899 tested  (41.2 %) and 1347861 untested  (58.8 %),
         0 have seen errors.
14 For tested frames, the following statistics have been calculated:
15   Time it took to claim a frame (in jiffies) (min,max,avg) : 0, 1, 0
16   Timestamp of last test (min,max,avg) : 2010-06-03 22:46:30, 2010-06-13 12:34:09, 2010-06-04 05:42:36
```

Listing 5.3: *Output Of the Scheduler*

This listing shows the output of the scheduler when it runs on a machine with 8 GiB of memory (2293760 frames of 4 KiB).

Lines 4–7, and 13–16 show statistics about the testing process. Remarkably is the nearly non-existent cost of frame acquisition (this test scheduler only tested free buddy pages).

In line 9 the test has been canceled by the user. Lines 12–16 show, that the scheduler implementation can transparently cope with restarts.

5.2 User Space

The user space contains both the frame scheduler, and the fault model-implementations. The implementation contains two different fault models (requirements 8+9), and two different scheduling strategies. The scheduling strategies mainly differ in the amount of frames to request from the kernel module.

Listing 5.4 shows a shortened implementation of a scheduler. The design presented in chapter 4 proved itself again by allowing a very easy and straightforward implementation. Extensive unit tests, and the creation of libraries supported this as well.

In fact, most of the implementation complexity has been moved into libraries, easing the implementation of new scheduling strategies. All in all, the python implementation consists of ca. 3000 lines of code, including ca. 800 lines of unit tests.

```
 1 from scheduling.helpers import is_free_buddy
 2 from scheduling.helpers import is_anon_page
 3 
 4 class SimpleBlockwiseScheduler(object):
 5     # ...
 6 
 7     # Main loop
 8     def run(self, first_frame,last_frame, allowed_sources):
 9 
10         max_blocksize = 100
11         max_non_matching = 100
12 
13         cur_non_matching = 0
14 
15         block = []
16 
17         for pfn in xrange(first_frame, last_frame):
18             frame_status = self._pfn_status(pfn)
19 
20             if ( self.should_test(frame_status)):
21                 block.append(frame_status)
22                 cur_non_matching = 0
23             else:
24                 cur_non_matching += 1
25 
26         if (len(block) > 0  ):
27                 self.test_frames_and_record_result(block, allowed_sources)
28 
29 
30 
31     # evaluate, if the frame should be tested now
32     def  should_test(self,frame_status):
33         now = self.timestamping.timestamp()
34         time_untested = now - frame_status.last_successfull_test
35         return  (time_untested > self.max_untested_age )
36 
37     def  test_frames_and_record_result(self,frame_stati, allowed_sources):
38         # ...
39         # Claim the memory
40         results = self._claim_pfns(pfns, allowed_sources)
41 
42         for frame in results:
43             if  frame.pfn in status_by_pfn:
44                 # ...
45                 if frame.is_claimed():
46                     # ...
47                     is_ok = False
48 
49                     with self.physmem_device.mmap(physmem.PAGE_SIZE * num_frames_claimed) as map:
50                         # It is better to handle bad frames after they are unmapped
51                         is_ok = self.frame_test.test(map,frame.vma_offset_of_first_byte  ,physmem.
                                 PAGE_SIZE)
52 
53                     if is_ok:
54                         # ...
55                         self._report_good_frame(frame.pfn)
56                     else:
57                         # ...
58                         self.physmem_device.mark_pfn_bad(frame.pfn)
59                         self._report_bad_frame(frame.pfn)
60     # ...
```

Listing 5.4: *Implementation of a Scheduler*

This listing shows the core functionality of a frame scheduler. The implementation presented here gathers batches of frames to be tested (lines 10–24) based on an evaluation function that determines, whether the frame should be tested or not (lines 32–35). Starting with line 37, the actual test is implemented. Line 51 shows the call to the strategy that implements the fault model, and line 58 shows how the scheduler reports a bad frame to the kernel module.

5.3 Implementing a Memory Tester for Linux

In section 2.4 it had been discussed, whether or not Linux is a suitable operating system to implement an online memory test. On the plus side for Linux are a sophisticated operating system with a broad user base, wide spread usage, and an open source license suitable for scientific publication. The lack of coherent documentation – especially for 64 bit kernels, where documentation is nearly non existent –, the lack of stable kernel interfaces, and the rapid and far-reaching changes made in the memory subsystem make development for the mm very difficult.

5.3.1 Difficulties

Matters are complicated further by the fact that the existing implementation and documentation cover the regular way of memory allocation: The client requests memory of a certain size and with certain properties, e.g. "allocate x-frames of memory in the DMA zone, and do not sleep". The mm allocates memory suitable for the clients request in a highly optimized way.

The requirements of a memory tester are very different: A memory tester needs to access each physical frame available in the system. To do so, the memory test needs to allocate memory not by size, but by frame.

Using the example of the simplified mm architecture in figure 2.6, the difference between the two needs can be visualised as *top down* versus *bottom up*. A regular memory allocation, e.g. by a process that needs more memory, traverses the software stack *top-to-down*.

Each layer uses the service of the layer below. In doing so, the ownership of memory, concretely the ownership of *struct page* instances, is changed. As soon, as the ownership of objects passed from the callee to the caller, the caller can use these objects as it needs. The caller can store references to the objects in the callers internal data structures, the caller can appoint specific contracts about handling these objects – without regard to the previous owner. In short, the lower layers of the software stack should not, and in most cases cannot, reclaim a specific frame because it often is not even clear, who currently owns a frame, let alone how the frame is used in the current owners internal structures.

A memory tester that needs to claim a *specific* frame needs to solve exactly these problems, it needs to traverse the software stack *bottom-up*. For example, if the memory tester wants to claim a specific frame that is known to be free via the Buddy Allocator, it needs to directly manipulate the internal data structures of the Buddy Allocator. In doing that, it has to account for the data structures themselves, and the dynamic behaviour of the Buddy Allocator, including threading and locking. Being thoroughly optimized for performance, especially the dynamic behaviour of the whole Linux kernel is very difficult to understand.

If the pace of changes to the Linux kernel, and the other aspects discussed in section 2.4.3 are taken into account, the only viable solution seems to be to rely on existing structures in the kernel, that – sometimes only partially – solve similar problems.

5.3.1.1 Stability of the HW-Poison Component

The HW-Poison component [79] within the Linux kernel handles memory hardware detected memory errors. It is sponsored by Intel [72], and a part of the Linux kernel since June 2009 [60].

A central part of its functionality is the ability to free frames currently used by various parts of the system. In accordance with requirement 25, the original concept based on the idea to utilize the HW-Posion component as a central acquisition strategy for the kernel module. Further strategies would be implemented to provide extensibility, and modularity (requirements 18+19). They would handle performance, and other issues found with the HW-Poison component.

Unfortunately, the HW-Poison component turned out to be unusable for the implementation. Despite being stated by its implementation documentation [79], it was impossible to return frames acquired by the HW-Poison component back to the system. In effect that meant, that using the HW-Poison component as strategy implementation would deplete the system of all usable memory. Additionally, using the HW-Poison strategy caused very frequent system lockups, and “mm-OOPS”.

In private communication, one of the authors of HW-Poison said, that it *should* work for a “standard user space page, no race (no ptrace, direct IO, fork, hard offline in parallel etc.)”, but unfortunately this did not work as of kernel 2.6.34.

Chapter 6

Evaluation

This chapter evaluates the implementation with respect to performance, and fulfillment of the requirements.

6.1 Performance

Evaluating the performance of a complex system is a complex endeavour in itself, and meaningless, if it is not clear what should be measured. The performance goals of the implementation are stated in the first chapter: To be able to estimate, if running memory tests on a production low-end server is feasible.

The performance metrics chosen for this diploma thesis are closely related to the applications most often found on the targeted class of machines. For this reason, the choice fell to measuring the performance of a complete application stack consisting of *MySql* [15], *apache2* [2], and the content management platform *Drupal* [3]. The application stack is tested by using *ab*, the apache HTTP server benchmarking tool [1].

Compiling the Linux kernel puts a heavy load on a system. Measuring the time it takes to compile the Linux kernel while the memory tester runs in the background is another benchmark measured for this diploma thesis.

6.1.1 Hardware

The system used for benchmarking had a AMD Athlon(tm) 64 X2 Dual Core Processor 4600+ with 2.4 Ghz installed. Memory wise the system had 8 GiB of DDR2 memory (800MHZ) installed.

6.1.2 Evaluating the Impact on a Web-Application Stack

The following test setup has been chosen: The Drupal installation uses a local MySql database for storage. The Drupal installation is filled with example data[1]. A remote hosts conducts load tests by running the *ab* benchmarking tool over gigabit ethernet.

The assumption is that the memory test reduces the performance of the application. To test this, the performance benchmark is run under three conditions.

The following variations were chosen for measurement:

No Test The benchmark is run *without* the memory tester running.

Low Priority The benchmark is run *with* the memory tester running with low priority (by running it with `nice 20`).

Normal Priority The benchmark is run *with* the memory tester running with normal priority.

The following performance metrics were chosen for measurement:

Requests/Second The number of requests served by the web-application per second.

Frames/Second The number of frames tested by the memory tester using the linear time complexity algorithm. Measured in frames per second.

The results of these tests can be seen in figure 6.1. It seems, that the running memory tester did have no impact on the performance of the web application.

[1] An archive with the used example data and application can be downloaded here: `http://drupal.org/node/664000`

Further investigation showed, that the application cached the result pages in memory, and this made it very resilient to the CPU-bound load of the memory test. To verify this theory, the test has been redone by replacing *ab* with the vulnerability scanner *skipfish* [34]. As a vulnerability scanner with a brute force approach, skipfish puts a much higher load on the system, and – more important – uses `HTTP POST` operations that are not cached by the application. The results ascertained by running skipfish were partly difficult to interpret. Due to the probabilistic nature of the skipfish tests, the performance metrics gained from the client side were not conclusive: The requests per second metric varied between 2417 and 3243, independently whether an instance of the test application ran or not. It could be measured though, that lowering the priority of the testing process has an affect on the number of frames scanned per second, while the scan ran: 4 frames/s for a process with normal priority, 0 frames/s for a process that has been "niced" by 20.

6.1.3 Evaluating the Impact on Kernel Compile Time

The following test setup has been chosen: The Linux kernel is compiled with the $-j2$ flag, instructing the compile chain to use both of available processor cores.

The assumption is that the memory test increases the time it takes to compile the Linux kernel.

The same variations as in the web application-case were chosen for measurement. The following performance metrics were chosen for measurement:

Seconds to Compile the Kernel The number of seconds it took to compile the kernel.

Frames/Second The number of frames tested by the memory tester using the linear time complexity algorithm. Measured in frames per second.

The results of these tests can be seen in figure 6.2. Running the memory test does have an impact on the kernel compile time. When the memory test is run with reduced priority, the kernel compile time is not affected to the same extend. The influence of the reduced process priority posed the question, what

caused the kernel compile time to be increased in the first place. Two possible factors were identified: Processor utilisation, and memory bandwidth.
To determine the influence of these two factors, two micro benchmarks were realised. The first benchmark compiled the kernel while a task put significant strain on the CPU by running in a tight loop. The second benchmark stressed the memory bus by continuously writing in a 16 MiB memory area.
The results of this benchmark can be seen in figure 6.3. Again, the priority of the stress test had a very significant influence on the performance of the kernel compilation. This can be explained by assuming that in this benchmark the CPU cycles are more scarce than memory bandwidth.

6.2 Requirements

The following sections discuss, in how far the implementation has met the stated requirements, and were is room for improvement.

6.2.1 Stability

Manipulating the memory management of a running kernel is a very risky thing to do, and for that reasons, system stability is an important topic to discuss. The implementation presented in this thesis has proven to run very stable by letting the memory tester run on a live webserver for more than three days without any stability problems. Said experiment ran solely with the Buddy-allocation strategy enabled. This strategy aims to fulfil requirement 5, and allows to acquire free buddy pages by manipulating the buddy allocators internal structures. The free page cache strategy, and the anonymous memory strategy lead to reproducible crashes an corruptions. Using the HW-Poison strategy lead to the same results.
On one hand, the overall stability of the system is to be judged positively, because the implementation ran for several days, and was only stopped for a scheduled downtime. On the other hand, the stability the important strategy components is imperfect. Despite this restriction, the implementation can be said to be very stable – among other things this can be attributed to strict

adherence to the *principle of least privilege*, and the schedulers ability to select the acquisition strategies.

Regarding the requirements stated in section 4.1 it can be said, that all stability related requirements (20–25) are met.

6.2.2 Memory Coverage

It is required, that the implementation should be able to test a substantial amount of the installed memory (requirement 3). The long term test mentioned in the previous section ran on a machine with 8 GiB of memory. At the end of the three days, the implementation had been able to test 68% (5.44 GiB) of all memory frames.

Regarding the requirements stated in section 4.1 it can be said, that not all memory coverage related requirements (3–7) are met. This can be explained by the extreme complexity and difficulty of interfering with the Linux mm. The stability issues that prevented the HW-poison patches from being used are an evidence for this complexity.

6.2.3 Error Detection

A core requirement is that the implementation must be able to detect memory errors. To reliably verify the detection capabilities of a memory tester it needs to run in an environment with known errors. This task can be greatly simplified by running the detection software in a simulated environment with fault injection capabilities. One such environment is the *FAUmachine* [100] developed at the Friedrich-Alexander-Universität Erlangen-Nürnberg. Using the FAUmachine all of the injected stuck-at faults have been found. The faults were carefully injected into locations known to be free, and testable by the implementation. This proved w.l.o.g. that the requirement of fault detection (requirement 1) is fulfilled. An inspection of the page-flags of the identified frame showed, that the `HW_POISON` flag had been set. This flag prevents the Linux mm to use this frame for further allocations, thus fulfilling the requirement of page isolation (2).

6.2.4 Visualisation

The implementations ability to visualize the memory usage has already been presented in section 4.2.5, thus the requirement of visualising the memory usage (requirement 15) is fulfilled.

6.2.5 Fulfilment of Requirements

Nearly all the 25 stated requirements have been fulfilled, only two requirements – reliably testing anonymous memory and frames in the page cache – are still too unstable to consider them fulfilled.

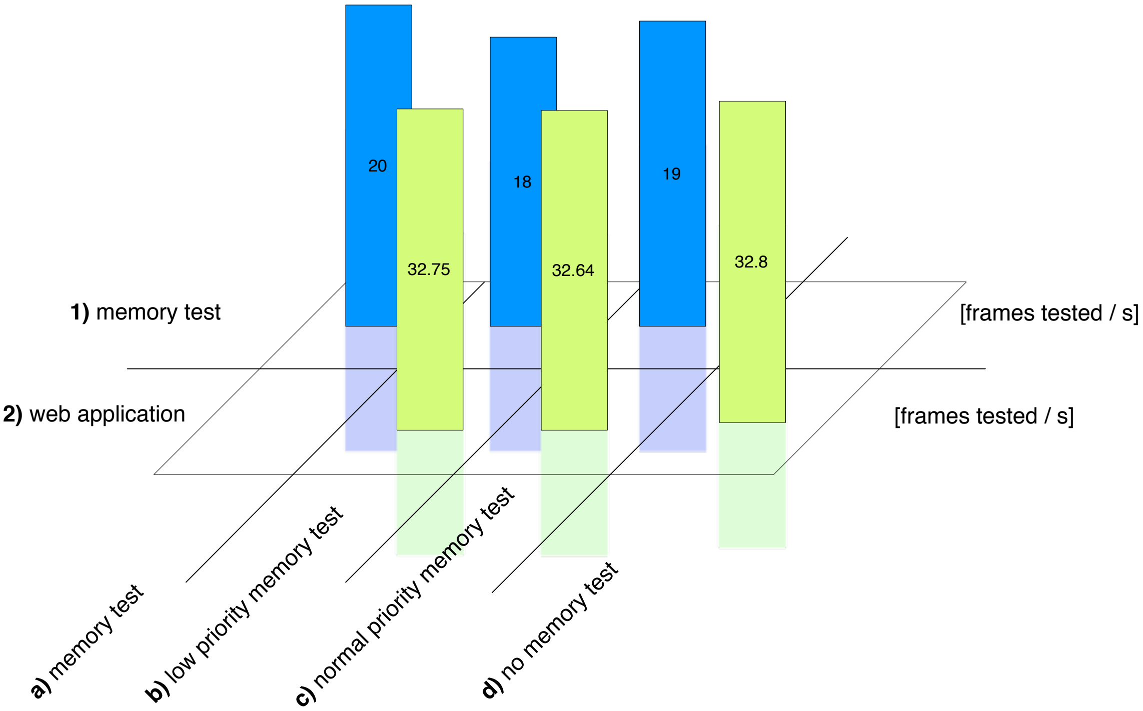


Figure 6.1: *Performance Evaluation: Web Application*
This figure show the results of a benchmark that simulates a webserver with mostly read-only accesses, which can be served mostly from a cache.

The chart shows how the performance of the memory tester (bars **1 a-c)**, blue) and the performance of the web-application (bars **2 b-d)**, green) varies under the four test scenarios.
The following test cases are depicted:

a) Running the memory test without accessing the web application.
b) Running the memory test with *low* priority while accessing the web application with *ab*.
c) Running the memory test with *normal* priority while accessing the web application with *ab*.
d) Accessing the web application with *ab*, without a running memory tester.

The *ab* tests where conducted with 5 concurrent sessions requesting a total of 20000 requests. Each test case was conducted five times. The numbers shown in the chart are the medians of the five measurements.

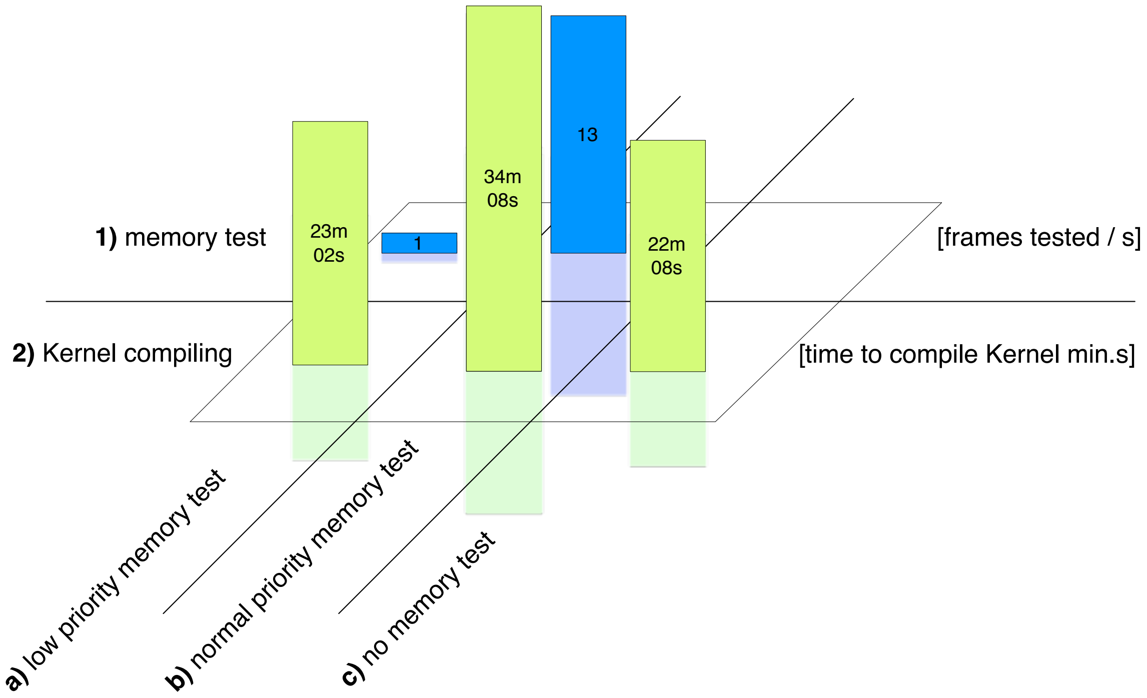


Figure 6.2: *Performance Evaluation: Kernel Compilation*
This figure shows how running a memory tester and compiling the Linux kernel affect each other.

The chart shows how the performance of the memory tester (bars **1 a-b**), blue) and the time it took to compile the kernel (bars **2 a-c**), green) vary under the three test scenarios.
The following test cases are depicted:

a) Running the memory test with *low* priority while compiling the kernel.
b) Running the memory test with *normal* priority while compiling the kernel.
c) Compiling the kernel without running a memory tester

Each test case was conducted five times. The numbers shown in the chart are the medians of the five measurements.

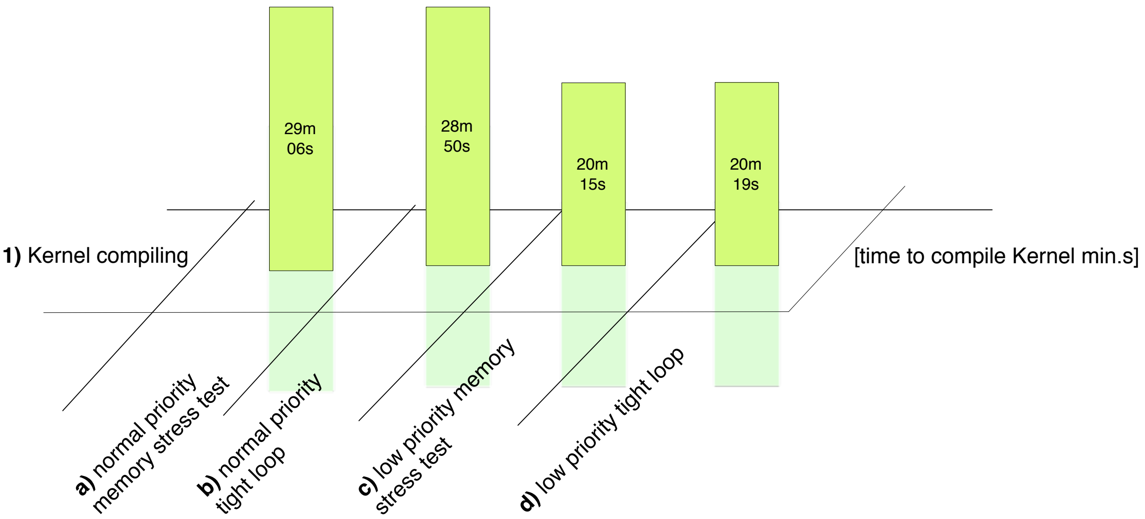


Figure 6.3: *Performance Evaluation: Kernel Compilation Micro Benchmark*
This figure shows how stressing the memory bus/the CPU affects the time it takes to compile the Linux kernel.

The chart shows kernel compilation time, while the system was put under stress at the same time.
The following test cases are depicted:

a) Running memory stress test with *normal* priority while compiling the kernel.
b) Running CPU stress test with *normal* priority while compiling the kernel.
c) Running memory stress test with *low* priority while compiling the kernel.
d) Running CPU stress test with *low* priority while compiling the kernel.

Each test case was conducted two times. The numbers shown in the chart are the average of the two measurements.

Chapter 7

Conclusion

This chapter concludes this diploma thesis, and answers the question, whether the author deems online memory tests for low- to mid-range servers to be feasible. The question will be answered by examining three aspects of the question: The technical feasibility, the impact on the systems performance, and the question, whether hardware based mechanisms would be better suited than a software implementation, or not.

This diploma thesis has proven that, technically, a software based memory scrubber is possible to implement. The immense amount of work and trial and error necessary to interfere with the internals of the Linux mm put this technical possibility into another light. Without a coordinated effort from the Linux kernel developer community it would be very difficult to extend the existing acquisition strategies, and – more important – keep them working correctly with the next kernel release. The current interest of the mm maintainers in memory fault handling (see section 3.2.3.1) could inspire further research in this area, and turn a technical possibility into a feasible endeavour.

Chapter 6 showed, that the performance impact of an online memory test depends on the workload of the system. For the targeted range of servers, especially those that mainly host non high-performance websites, the impact of an online memory test is tolerable to negligible.

This is an surprising result, because discussions privately held with kernel developers showed, that they thought that this kind of test would have a significant impact on the system performance.

The question, if a software based memory test is better than hardware based ECC cannot be answered with a single yes or no. Memory subsystems equipped with ECC provide a basic level of protection that greatly increases system reliability. Further, ECC subsystems can actively *correct* detected errors, something that cannot easily be done by a software based memory test. Despite these advantages of hardware based mechanisms, a software based mechanism has its own advantages. Software based mechanisms are far less restricted in the fault models they support. Standard SEC-DED ECC simply cannot detect a three bit error, a software based test can detect three bit errors, and many more. In theory, a software based test can implement complex fault models like the one proposed by Hayes (section 2.2).

Hybrid systems like virtualized ECC (section 3.3.2) can unify both worlds. But until these techniques reach the mainstream market, a software based memory test can still increase the reliability of RAM by detecting (complex) errors as soon as possible.

7.1 On the Feasibility of Modifying the Linux mm

A lesson learned from the experience of working with the Linux mm is, that the mm is a complex system with a lot of global state, fine grained synchronization, and very little coherent documentation. This can partially be attributed to the small group of people responsible for crafting the mm. Highly specialised developers and researchers working in an area of the kernel that was known for its difficulty even in the kernel 2.4 days [62] lead to an environment, where one *just knows*, which lock relates to which tasks, and who modifies what data structure under which circumstances. The stark focus on tuning caching, TLBs, and performance in general further complicates the comprehension of the static and dynamic behaviour of the mm.

The author of this diploma thesis has the notion that the mm would be much more accessible and malleable, if these implicit dynamic and locking dependencies would be reified as comprehensible aspects.

7.1.1 Adapting the System to Other Architectures

The memory problems described in this thesis are not restricted to systems running Linux. It would be interesting to compare the effort of implementing a memory test on a micro kernel architecture, e.g. L^4, to the effort of implementing the Linux pendant. The greatly reduced memory footprint of a micro kernel would allow a much greater test coverage of testable memory. Additionally the test overhead is probably reduced, because micro kernels generally define much cleaner and stricter interfaces, than conventional monolithic kernels.

Following the concept of micro kernels to kernels with a small core, and very stringent boundaries leads to hypervisors. Memory reliability, and testing gets more important, when more services are provided by single hardware systems. Hypervisors like VMWare, Xen, or KVM are used to consolidate many physical machines into virtualized machines. This concentration leads to single points of failure that potentially affect a large number of services.

Contrary to consolidation servers are the myriad embedded systems used in modern society. Verifying the functional behaviour of a devices memory can – depending on the operating range of the devices – be of even more importance, than verifying the memory of a server in a controlled environment. Implementing software based memory tests in embedded devices could eventually lead to improved reliability, and safety. By implementing test algorithms in software, the hardware complexity of programmable devices can be reduced, and more flexible and more complex algorithms can be implemented.

By separating most of the concerns into dedicated modules, and by adhering to the principle of least privilege, the design presented in this thesis perfectly fits the micro-kernel idea. Merging layers together, as it may be needed to apply the design to embedded systems, is always easier than splitting up a layer, thus the design should be equally feasible for more complex embedded systems.

7.1.2 Design Improvements

Although the design proved itself, a few modifications and extensions could improve the design. The biggest drawback of the current design is, that the strategies used to acquire memory work on a frame-by-frame base. By allowing the strategies to allocate more frames in one step, the amount of lock acquisitions could be reduced.

Another improvement could be a strategy that allocates frames by directly calling into the memory management. Taking Linux as an example, a strategy that acquires 2^n frames from the buddy allocator does not need to dangerously manipulate kernel data structures.

Implementing adaptive test strategies is another possible direction for future research. Adaptive strategies would choose more complex fault models, and more aggressive acquisition strategies, for memory modules where memory errors are suspected.

Fortunately, none of these improvements invalidates the general design. Especially the stringent split between complex algorithms running in user space, and (mostly) simple algorithms running in kernel space is not threatened by these improvements.

7.1.3 Closing Remarks

This diploma thesis is concluded by citing an announcement from NASA. The announcement is a very vivid example for the fallibility of RAM – even, when it is high quality material fit for leaving our solar system.

From the NASA website, updated May 17, 2010 at 5:00 PT:

> One flip of a bit in the memory of an onboard computer appears to have caused the change in the science data pattern returning from Voyager 2, engineers at NASA's Jet Propulsion Laboratory said Monday, May 17. A value in a single memory location was changed from a 0 to a 1. [52]

Appendix A

Requirements

Appendix B

Glossary, Acronyms & Symbols

Glossary

address decoding fault [i, 18]
Instead of cell C_i another cell C_j is addressed [95]. See section 2.2.3.

address space [i, 37, 44]
An address space is a range of addresses that have a specific meaning. In this thesis the term *address space* is used to describe the mapping of (virtual) addresses to physical addresses. See section 2.4.2.2.

buddy allocator [i, 37, 76]
The Buddy Allocator is the low level memory allocator of Linux kernel . See section 2.4.2 and [108].

Correctable Error (CE) [i, 5, 32, 33, 62, 69, 126]
An error that can be detected and corrected, see also UE and ECC.

checking sequence [i, 20, 21, 123]
"An input sequence that distinguishes a given n-state machine M from all other machines with the same input and output alphabets and at most n-states is called a checking sequence for M" [65]. Checking sequences are used to detect misbehaving memory.
See section 2.2.1 for a discussion.

ChipKill [i, 70–73]
An extension to ECC that allows the correction and detection of defect symbols (multiple bits). See section 3.3.1 for a discussion.

closed neighbourhood [i, 20]
A neighbourhood N is said to be *closed*, when it is a partition of M_r:
$\forall N_i, N_j \in N : \forall C_i \in N_i : C_i \in N_j \Leftrightarrow N_j = N_i$ [65].
See section 2.2.1 for a discussion.

compound page [i]
A compound page consists of several – 2^n, where valid numbers for n depend on the processor – consecutive frames. This grouping allows an optimisation where one compound page needs only one instead of 2^n entries in the TLB. This improvement allows more pages to be covered by TLB entries, thus lowering the chance for expensive TLB-misses.
The Linux kernel uses large pages for memory reserved for the kernel. Talluri and Hill discussed the benefits of compound pages – they use the name “superpages” – in their 1994 work [116].

coupling fault [i, 18, 22, 122, 127]
A type of fault, where the change of one memory cell incorrectly changes another memory cell. This is the same as a 2-coupled fault [95]. See k-coupled fault, section 2.2.3.

Dual Inline Memory Module (DIMM) [i, 10, 42, 69]
Computer memory is typically installed as separate modules. A widespread type of modules are DIMM modules, because they are used for DDR-SDRAM memory.

Desktop Management Interface (DMI) [i, 49]
“DMI generates a standard framework for managing and tracking components in a desktop pc, notebook or server. DMI was the first desktop management standard. The DMI Home Page is a repository of all DMI-related information from the specification to tools to support to the Product Registry of DMI-certified products. Due to the rapid advancement of DMTF technologies,

such as CIM, the DMTF defined an "End of Life" process for its Desktop Management Interface (DMI), which ended March 31, 2005." [25] DMI is still used by Linux, for example by the mcelog package. See also mcelog.

e820 map [i, 44]
The e820 map can be queried from the BIOS of x86-based computers and contains the memory map for this system [54].

Error Correction Codes (ECC)[i, 5, 6, 8–10, 29, 31, 32, 54, 58, 68, 70, 72, 114, 119, 121, 126]
A hardware extension, where additional data is used to provide error detection and correction for memory devices. Usually 64 bit words are protected by an SEC-DED -scheme [83]. For more details be referenced to [64, 85, 50, 88]

Error Detection And Correction (EDAC) [i, 8, 60, 68, 121]
The EDAC project is an extension of the Linux kernel and contributes a set of Linux kernel modules for handling hardware-related errors. Its major focus has been ECC memory error handling, however the modules also detect and report PCI bus parity errors. [4]

Eversholt [i, 66, 67]
A language designed to describe fault trees. See section 3.2.1.3 and the Eversholt manual [114].

fault model [i, 11, 13, 15, 20–23, 26, 59, 75–78, 80, 82, 83, 85, 87, 88, 95, 97, 98, 114]
A fault model describes types of faults. Specifying a fault model allows the prediction of the consequences of faults, and the design and verification of fault detection and prevention measures. See section 2.2.1 for a discussion.

fault tree [i, 66, 121]
Fault trees graphically represent how faults and other events interact within a system. See sections 2.3.2, section 3.2.1.3, and the Eversholt manual [114]. A basic introduction into fault tree analysis can be found in [63]. Ramamoorthy *et al.* published a paper that presents how Fault Tree Analysis can be applied

to faults in computer systems [101].

Failures In Time per billion device hours (FIT) [i, 14, 15, 122]
The FIT-rate is used to describe the reliability of (semiconductor) elements and is defined as $FITrate = \frac{Pn}{DeviceHours * AF} * 10^{-9}$, where Pn and AF are used for accelerated stress testing. AF is the acceleration factor between stress environment and typical operating conditions, and Pn = Poisson statistic at 60% confidence [86].

Fault Management Resource Identifier (FMRI) [i]
A unique identifier for an entity that is managed or observed by the Solaris Fault Management, see also SMF.

frame [i, 14, 37–40, 47, 59, 84, 99, 136]
Most processors and operating systems manage memory in fixed units called frames. The commonly used x86 architecture supports frame sizes of 4 KiB and 2 or 4 MiB. The usage of the word frame generally means that physical memory is addressed without address translation via the TLB.

hard error [i, 14, 70]
Errors that are caused by physical defects are called hard errors [64, 86].

***k*-coupled fault** [i, 23, 120, 122]
A k-coupling fault is a fault, where the change of one cell C_i in a set of k cells C^k causes a change in $C_j \in C^k$ when the other $k-2$ cells in C^k have some fixed values. See coupling fault, and section 2.2.3.

***k*-coupled fault, restricted** [i, 23]
A restricted k-coupled fault is a $k-$coupled fault with non-overlapping k-sets [95].

Linux [i, 1, 2, 8–11, 13, 14, 16, 33–36, 38–40, 43–45, 47–51, 53, 54, 57–60, 67–70, 76, 77, 81, 83, 84, 91, 93, 99, 100, 103, 105, 107, 110, 111, 113–116, 119–121, 126, 127, 136]
“Linux is a clone of the operating system Unix, written from scratch by Linus

Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX and Single UNIX Specification compliance. It has all the features you would expect in a modern fully-fledged Unix, including true multitasking, virtual memory, [. . .], proper memory management, [. . .]. Although originally developed first for 32-bit x86-based PCs (386 or higher), today Linux also runs on (at least) [. . .] AMD x86-64, [*and many more*] architectures; for many of these architectures in both 32- and 64-bit variants. Linux is easily portable to most general-purpose 32- or 64-bit architectures as long as they have a paged memory management unit (PMMU) [. . .]" [12]

local access [i, 42]

In NUMA systems, if a processor accesses memory on its own node, this is called a local access (compare to [**?**]).

Local Pattern Sensitive Fault (LPSF) [i, 20]

Another name for NPSF.

Machine Check Exception (MCE) [i, 29, 36, 53, 69]

Intel x86 processors signal certain error conditions by raising so called Machine Check Exceptions as interrupts [9].

mcelog [i, 8, 60, 121]

"mcelog decodes machine check events (hardware errors) on x86-64 machines running a 64-bit Linux kernel." [30]

memory tester [i, 1, 6, 7]

A memory tester aims to verify the correct behaviour of RAM by writing and verifying checking sequences to the memory.

misbehaving [i]

The word *misbehaving* is used to describe a service or component that violates the contract with its clients.
See section 1.1 for a discussion.

neighbourhood [i, 20, 124]

A neighbourhood is a set N of sets N_k, where N_k consists of memory cells. The neighbourhood of a memory cell C_i is called N_i. N_i contains C_i [65].
See section 2.2.1 for a discussion.

Neighbourhood Pattern Sensitive Fault (NPSF) [i, 78, 123, 124, 126]
A NPSF is a restriction of the PSF, where a fault is local to a given neighbourhood N_i.
See also SPSF and PSF, also see section 2.2.1 for a discussion.

Non Uniform Memory Access (NUMA) [i, 10, 37, 38, 42, 43, 45, 48, 57, 123]
NUMA is a computer memory design used in multiprocessor systems. The memory access time in NUMA systems depends on the memory location relative to a processor – a processor can access its own local memory faster than non-local memory [23].

open neighbourhood [i, 20, 21]
A neighbourhood N is said to be *open*, when it is no partition of M_r.
See section 2.2.1 for a discussion.

page [i, 39, 40, 59]
A *page* is a *frame* that is mapped into user or kernel space.

page fault [i, 40]
A page fault happens, when a process accesses a page that is either not connected to a physical frame, or if the process accessed the page with an invalid operation, e.g. writing to a read-only page. See section 2.4.2.2.

physical address [i, 44, 59, 126]
The physical address is the adddress send to the memory bus in order to access a particular storage cell in the RAM.
Compare to virtual address.

Pattern Sensitive Fault (PSF) [i, 13, 18, 20, 21, 124, 126]
A very general fault type that, for an r-bit memory, can be detected by an

$O\left(\left(3r^2 + 2r\right) 2^r\right)$ algorithm [65].
See also NPSF, SPSF, and section 2.2.1 for a discussion.

Random Access Memory (RAM) [i, 13, 15, 16, 19, 23, 38, 40, 42, 44, 46, 53, 54, 57, 68, 77, 114, 116, 123, 124]
Hayes defined *memory* as a set of r addressable binary storage cells $M_r := \{C_0, C_1, \ldots, C_{r-1}\}$, where i denotes the address of C_i and where each storage cell C_i can store exactly one bit. Operations $READ$ and $WRITE$ can be performed on M_r, each cell can be written to or read from independently of previous $READ$ or $WRITE$ operations [65]. Modern forms of RAM, like DDR-SDRAM are no longer bit-addressable but word-addressable. RAM is also called system memory or just memory.
See section 2.2.1 for a discussion.

Single Error Correction – Doubble Error Detection (SECDED) [i, 5, 70, 71, 114, 121]
An error detection and correction scheme that allows the detection of two bit errors and the correction of one bit errors per memory word [83, 50, 67].

slab allocator [i, 37]
The Slab-Allocator manages finer grained memory allocations. See section 2.4.2 and [46, 62].

Service Management Facility (SMF) [i, 33, 62–64, 67, 122]
"SMF is designed to simplify the management of system and application services. It delivers new and improved ways to control services, and tries to restart failed services automatically. In addition, SMF allows administrators to define the relationships between services. It is now possible to define a service that is dependent on other services – a dependent service will not run unless the other services that it requires are already running. Through a set of new administrative interfaces, SMF allows services to be easily and consistently configured, enabled, and controlled, at the same time providing better visibility of errors and improved debugging capabilities to resolve service-related problems quickly when they occur." [104]

See section 3.2.1 for a discussion.

soft error [i, 14]
Errors that are caused by external factors like α-radiation or temperature are called soft errors [115, 71].

Solaris [i, 6, 11]
Initially developed by Sun, Solaris is a UNIX operating system. Solaris 10 implements a FMA that is discussed in section 2.3.

Single Pattern Sensitive Fault (SPSF) [i, 21, 124]
A restricted PSF where only one memory cell misbehaves and the errnous interaction between memory cells is restricted to α distinct neighbourhoods, each with the length q. All SPSFs for an r-bit memory can be detected by an $O\left((4q+3)\,2^q * r\right)$ algorithm [65].
See also NPSF, and PSF, also see section 2.2.1 for a discussion.

struct page [i, 47]
A *frame* is a hardware supported construct. A *page* is a software construct. The Linux kernel manages physical memory by keeping an instance of *struct page* for each physical frame.

stuck-at fault [i, 18, 22, 107, 127]
A cell C_i is said to suffer from a stuck-at 0 or stuck-at 1 fault, when the content of C_i cannot be changed: C_i is stuck at x when $W_iR_i = \bar{W}_iR_i = x$ [95]. See section 2.2.3.

Translation Lookaside Buffers (TLB) [i, 46, 78, 114, 120, 122, 126]
The TLB is used on the x86 architecture to translate virtual addresss into physical addresss [72].

Uncorrectable Error (UE) [i, 5, 32, 58, 69, 119]
An error that can be detected, but not corrected, see also CE and ECC.

virtual address [i, 44, 59, 124, 126]
Virtual addresses are used to abstract from physical addresss. The mapping

from virtual address to physical address is done by the MMU. Most operating system, including Linux, use virtual addressing for kernel- and user-space processes.

writer logic [i]

According to Nair *et al.*, the Reader/Writer logic consists of the sense amplifiers, the write drivers and other supporting logic. They visualize stuck output lines of the sense amplifiers or writer drivers as stuck-at faults and shorts, or capacitive coupling between data input or output lines as coupling faults [95]. See section 2.2.3.

xVM [i, 64]

xVM is a port of the Xen hypervisor to the Solaris operating system [36].

zone [i, 44, 48]

Not all physical memory can be used for all purposes. Restriction apply for example to memory used for DMA, or for memory addressable by 32-bit kernels. Section 2.4 gives an overview of memory handling in Linux, and [62, 89, 53] cover the topic in more depth.

Symbols

C_i	The binary memory cell C_i.	i, 17, 19, 20, 22, 126
M_r	A random access memory of r binary cells $C_{0 \cdots r-1}$.	i, 17–21, 129
M_r^F	A faulty random access memory of r binary cells $C_{0 \cdots r-1}$.	i, 18–20
R_i	Read the content of cell C_i. Definition in 2.3.	i, 17–19, 22, 126
X_i	$X_i :=$ Either R_i, or W_i, or $\bar{W}_i\}$.	i, 17, 20, 22
$\tilde{W}_i$	$\tilde{W} :=$ Either W_i, or $\bar{W}_i$.	i, 17, 18
$\bar{W}_i$	Write a 0 into cell C_i. Definition in 2.2.	i, 17–19, 22, 126
W_i	Write a 1 into cell C_i. Definition in 2.1.	i, 17–19, 22, 126
z^*	$z^*(S, Y_j)$ is the output of the Mealy automaton, when the input series S is applied to M_r in state Y_j. The output is pruned of "—"s, so it contains only the output of $READ$s, i.e. $z^*(W_0 R_0 \bar{W}_1 R_1 R_0) = 101$.	i, 18, 22

Acronyms

BIOS	Basic Input Output System	i, 43, 49, 57
CE	Correctable Error	i, 119
CS	Chip Select	i, 68
DDR-SDRAM	Double Data Rate Synchronous Dynamic Random Access Memory	i, 40, 120, 125
DFG	Deutsche Forschungsgemeinschaft	i, 9
DIMM	Dual Inline Memory Module	i, 120
DMA	Direct Memory Access	i, 44, 45, 56, 99, 127
DMI	Desktop Management Interface	i, 120
ECC	Error Correction Codes	i, 121
EDAC	Error Detection And Correction	i, 121
EDP	Energy Delay Product	i, 73, 80
FIT	Failures In Time per billion device hours	i, 14, 15, 122
FMA	Fault Management Architecture	i, 13, 25, 29, 32, 33, 61, 62, 67, 68, 126
FMRI	Fault Management Resource Identifier	i, 62, 122

GART	Graphics Address Remapping Table	i, 46
GPL	GNU Public License	i, 35, 54
ISA	Industry Standard Architecture	i, 43
ITRS	International Technology Roadmap for Semiconductors	i, 9
KVM	Kernel Virtual Machine	i, 40
LPSF	Local Pattern Sensitive Fault	i, 123
MCE	Machine Check Exception	i, 123
mm	memory management	i, 11, 33, 34, 36, 45–48, 50–52, 76–78, 81, 83, 84, 89, 99, 100, 107, 113, 114
MMU	Memory Management Unit	i, 39, 126
NPSF	Neighbourhood Pattern Sensitive Fault	i, 124
NSA	National Security Agency	i, 34
NUMA	Non Uniform Memory Access	i, 42, 124
PCI	Peripheral Component Interconnect	i, 43

PFN	Page Frame Number	i, 14, 45
PSF	Pattern Sensitive Fault	i, 124
RAM	Random Access Memory	i, 125
SECDED	Single Error Correction – Doubble Error Detection	i, 125
SERD	Soft Error Rate Discrimination	i, 67
SMF	Service Management Facility	i, 125
SMP	Symmetric Multi-Processing	i, 40
SPSF	Single Pattern Sensitive Fault	i, 126
TLB	Translation Lookaside Buffers	i, 126
UE	Uncorrectable Error	i, 126
UMA	Uniform Memory Access	i, 38, 40, 42, 43

Appendix C

Extended listings

Some listings are too long to be placed embedded in the text. These listings follow on the next pages.

C.1 Listings for Chapter 2

```
/*
 * Each physical page in the system has a struct page associated with
 * it to keep track of whatever it is we are using the page for at the
 * moment. Note that we have no way to track which tasks are using
 * a page, though if it is a pagecache page, rmap structures can tell us
 * who is mapping it.
 */
struct page {
  unsigned long flags;    /* Atomic flags, some possibly
           * updated asynchronously */
  atomic_t _count;    /* Usage count, see below. */
  union {
    atomic_t _mapcount; /* Count of ptes mapped in mms,
           * to show when page is mapped
           * & limit reverse map searches.
           */
    struct {    /* SLUB */
      u16 inuse;
      u16 objects;
    };
  };
  union {
      struct {
    unsigned long private;    /* Mapping-private opaque data:
             * usually used for buffer_heads
             * if PagePrivate set; used for
             * swp_entry_t if PageSwapCache;
             * indicates order in the buddy
             * system if PG_buddy is set.
             */
    struct address_space *mapping;  /* If low bit clear, points to
             * inode address_space, or NULL.
             * If page mapped as anonymous
             * memory, low bit is set, and
             * it points to anon_vma object:
             * see PAGE_MAPPING_ANON below.
             */
      };
#if USE_SPLIT_PTLOCKS
      spinlock_t ptl;
#endif
      struct kmem_cache *slab;  /* SLUB: Pointer to slab */
      struct page *first_page;  /* Compound tail pages */
  };
  union {
    pgoff_t index;    /* Our offset within mapping. */
    void *freelist;   /* SLUB: freelist req. slab lock */
  };
  struct list_head lru;   /* Pageout list, eg. active_list
           * protected by zone->lru_lock !
           */
 // [ ... lines removed for brevity ... ]
#if defined(WANT_PAGE_VIRTUAL)
  void *virtual;      /* Kernel virtual address (NULL if
             not kmapped, ie. highmem) */
#endif /* WANT_PAGE_VIRTUAL */
#ifdef CONFIG_WANT_PAGE_DEBUG_FLAGS
  unsigned long debug_flags;  /* Use atomic bitops on this */
#endif

#ifdef CONFIG_KMEMCHECK
  /*
   * kmemcheck wants to track the status of each byte in a page; this
   * is a pointer to such a status block. NULL if not tracked.
   */
  void *shadow;
#endif
};
```

Listing C.1: *struct page*

This is an excerpt from `linux/mm_types.h` and contains the definition of *struct page*, a structure central to frame-management in the Linux kernel. The extensive use of *defines* and *unions* make the binary representation of a *struct page* instance found in memory difficult to analyze. See section 2.4 for details, and listing C.2 for a valid values for *flags*.

```
/*
 * Various page->flags bits:
 *
 * PG_reserved is set for special pages, which can never be swapped out. Some
 * of them might not even exist (eg empty_bad_page)...
 [ ... lines removed for brevity ... ]
 *
 * PG_highmem pages are not permanently mapped into the kernel virtual address
 * space, they need to be kmapped separately for doing IO on the pages.  The
 * struct page (these bits with information) are always mapped into kernel
 * address space...
 *
 * PG_buddy is set to indicate that the page is free and in the buddy system
 * (see mm/page_alloc.c).
 *
 * PG_hwpoison indicates that a page got corrupted in hardware and contains
 * data with incorrect ECC bits that triggered a machine check. Accessing is
 * not safe since it may cause another machine check. Don't touch!
 *
  [ ... lines removed for brevity ... ]
 * The page flags field is split into two parts, the main flags area
 * which extends from the low bits upwards, and the fields area which
 * extends from the high bits downwards.
 [ ... lines removed for brevity ... ]
 *
 * The fields area is reserved for fields mapping zone, node (for NUMA) and
 * SPARSEMEM section (for variants of SPARSEMEM that require section ids like
 * SPARSEMEM_EXTREME with !SPARSEMEM_VMEMMAP).
 */
enum pageflags {
  PG_locked,    /* Page is locked. Don't touch. */
  PG_error,
  PG_referenced,
  PG_uptodate,
  PG_dirty,
  PG_lru,
  PG_active,
  PG_slab,
  PG_owner_priv_1,  /* Owner use. If pagecache, fs may use*/
  PG_arch_1,
  PG_reserved,
  PG_private,   /* If pagecache, has fs-private data */
  PG_private_2,   /* If pagecache, has fs aux data */
  PG_writeback,   /* Page is under writeback */
#ifdef CONFIG_PAGEFLAGS_EXTENDED
  PG_head,    /* A head page */
  PG_tail,    /* A tail page */
#else
  PG_compound,    /* A compound page */
#endif
  PG_swapcache,   /* Swap page: swp_entry_t in private */
  PG_mappedtodisk,  /* Has blocks allocated on-disk */
  PG_reclaim,   /* To be reclaimed asap */
  PG_buddy,   /* Page is free, on buddy lists */
  PG_swapbacked,    /* Page is backed by RAM/swap */
  PG_unevictable,   /* Page is "unevictable"  */
#ifdef CONFIG_MMU
  PG_mlocked,   /* Page is vma mlocked */
#endif
#ifdef CONFIG_ARCH_USES_PG_UNCACHED
  PG_uncached,    /* Page has been mapped as uncached */
#endif
#ifdef CONFIG_MEMORY_FAILURE
  PG_hwpoison,    /* hardware poisoned page. Don't touch */
#endif
  //
//  [ ... lines removed for brevity ... ]
  /* SLUB */
  PG_slub_frozen = PG_active,
  PG_slub_debug = PG_error,
};
```

Listing C.2: *Flags for struct page*

This is an excerpt from `linux/page-flags.h` and contains the definition of the flags for *struct page.flags*. Noteable is, that some flags have a different meaning, depending on which subsystem the frame is allocated to. See section 2.4.

Bibliography

[1] *ab - Apache HTTP server benchmarking tool.* `http://httpd.apache.org/docs/2.0/programs/ab.html`

[2] *The Apache Software Foundation.* `http://www.apache.org/`

[3] *Drupal homepage.* `http://drupal.org/`

[4] *EDAC website.* `http://bluesmoke.sourceforge.net/`

[5] *Homepage of the gentoo Linux distribution.* `http://www.gentoo.org/`

[6] *Homepage of the RedHat Linux distribution.* `http://www.redhat.com/`

[7] *Homepage of the SuSe Linux distribution.* `http://www.suse.com/`

[8] *Homepage of the Ubuntu Linux distribution.* `http://www.ubuntu.com/`

[9] Intel: *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Volume 3A: System Programming Guide, Part 1. Order Number: 253668-034US : `http://download.intel.com/design/processor/manuals/253668.pdf`,

[10] *Linux Kernel Documentation: NO-MMU MEMORY MAPPING SUPPORT.* [Kernel Source 2.6.34RC4]/Documentation/nommu-mmap.txt,

[11] *Linux Kernel Documentation: Pagemap.* [Kernel Source 2.6.34RC4]/Documentation/vm/pagemap.txt,

[12] *Linux Kernel homepage.* `http://www.kernel.org/`

[13] *Memtest86+.* `http://www.memtest.org/`

[14] *Memtest86 algorithms.* `http://www.memtest86.com/tech.html#algo`, Abruf: 2010-04-15

[15] *MySql Homepage.* `http://www.mysql.com/`

[16] *Python – C API Reference Manual.* `http://docs.python.org/c-api/`, Abruf: 2010-06-05

[17] *Python – Memory-mapped file support.* `http://docs.python.org/library/mmap.html`, Abruf: 2010-06-05

[18] *Python – The fcntl() and ioctl() system calls.* `http://docs.python.org/library/fcntl.html`, Abruf: 2010-06-05

[19] *Python homepage.* `http://www.python.org/`

[20] *Solaris 10 What's New: Predictive Self-Healing.* `http://docs.sun.com/app/docs/doc/817-0547/esqej`, Abruf: 2010-04-15

[21] *SWIG and Python.* `http://www.swig.org/Doc1.3/Python.html`, Abruf: 2010-06-05

[22] *Writing Device Drivers.* `http://docs.sun.com/app/docs/doc/819-3196/gemgv`, Abruf: 2010-04-15

[23] *Linux Scalability for Large NUMA Systems.* Proceedings of the Linux Symposium, 07 2003

[24] *High Memory In The Linux Kernel.* `http://www.ktrap.org/node/2450`. Version: 02 2004, Abruf: 2010-04-01

[25] *Desktop Management Interface (DMI) Standards.* `http://www.dmtf.org/standards/dmi/`. Version: 03 2005, Abruf: 2010-04-01

[26] *Virtual Memory on linux-mm.org.* `http://linux-mm.org/VirtualMemory/`. Version: 05 2006-05, Abruf: 2010-04-01

[27] *Linux Kernel Development.* `http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php`. Version: 04 2008, Abruf: 2010-04-01

[28] http://effogpled.googlecode.com/files/EffoDesign_MemTest.pdf

[29] *EDAC website.* http://pyropus.ca/software/memtester/. Version: 2009, Abruf: 2010-05-20

[30] *Homepage of the mcelog programm.* http://freshmeat.net/projects/mcelog. Version: 12 2009, Abruf: 2010-04-15

[31] *Hardware Error Kernel Mini-Summit.* http://events.linuxfoundation.org/lfcs2010/edac. Version: 05 2010, Abruf: 2010-05-20

[32] *Hardware Error Kernel Mini-Summit protocoll.* http://kerneltrap.org/mailarchive/linux-kernel/2010/5/17/4571295. Version: 05 2010, Abruf: 2010-05-20

[33] *Homepage of Distrowatch.* http://distrowatch.com/. Version: 05 2010, Abruf: 2010-05-20

[34] *Homepage of skipfish.* http://code.google.com/p/skipfish/. Version: 05 2010, Abruf: 2010-05-20

[35] *Project page of the "Upstart" project.* http://upstart.ubuntu.com/. Version: 2010, Abruf: 2010-04-15

[36] *Sun xVM Hypervisor Homepage.* http://hub.opensolaris.org/bin/view/Community+Group+xen/. Version: 2010, Abruf: 2010-04-15

[37] AL., NR A.: An overview of the BlueGene/L Supercomputer. In: *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing.* Los Alamitos, CA, USA : IEEE Computer Society Press, 2002, S. 1–22

[38] ALVES, Alexandre ; ARKIN, Assaf ; ASKARY, Sid ; BLOCH, Ben ; CURBERA, Francisco ; GOLAND, Yaron ; KARTHA, Neelakantan ; STERLING ; KÖNIG, Dieter ; MEHTA, Vinkesh ; THATTE, Satish ; RIJN, Danny

van d. ; Yendluri, Prasad ; Yiu, Alex: *Web Services Business Process Execution Language Version 2.0.* OASIS Committee Draft, May 2006

[39] AMD: *Performance Guidelines for AMD AthlonTM 64 and AMD OpteronTM ccNUMA Multiprocessor Systems.* Publication No. 40555, Revision: 3.00. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40555.pdf, june 2006

[40] AMD: *BIOS and kernel developer's guide for AMD NPT family 0Fh processors.* http://support.amd.com/us/Processor_TechDocs/32559.pdf, 07 2007

[41] Norm ANSI/IEEE 730 1989. *Software Quality Assurance Plans*

[42] Arcangeli, Andrea ; Eidus, Izik ; Wright, Chris: *Increasing memory density by using KSM.* http://www.kernel.org/doc/ols/2009/#19-28. Version: 2009

[43] Bairavasundaram, Lakshmi N. ; Arpaci-Dusseau, Andrea C. ; Arpaci-Dusseau, Remzi H. ; Goodson, Garth R. ; Schroeder, Bianca: An analysis of data corruption in the storage stack. In: *Trans. Storage* 4 (2008), Nr. 3, S. 1–28. http://dx.doi.org/10.1145/1416944.1416947. – DOI 10.1145/1416944.1416947. – ISSN 1553–3077

[44] Bligh, Martin J. ; Dobson, Matt ; Hart, Darren ; Huizenga, Gerrit: Linux on NUMA Systems. In: *Proceedings of the Linux Symposium* Bd. I, 2004, S. 90–102

[45] Borri, Simone ; Hage-Hassan, Magali ; Dilillo, Luigi ; Girard, Patrick ; Pravossoudovitch, Serge ; Virazel, Arnaud: Analysis of Dynamic Faults in Embedded-SRAMs: Implications for Memory Test. In: *J. Electron. Test.* 21 (2005), Nr. 2, S. 169–179. http://dx.doi.org/10.1007/s10836-005-6146-1. – DOI 10.1007/s10836–005–6146–1. – ISSN 0923–8174

[46] Bovet, Daniel P. ; Cesati, Marco: *Understanding the Linux Kernel, 3rd Edition.* O'Reilly, 2005

[47] Broy, Manfred ; Jarke, Matthias ; Nagl, Manfred ; Rombach, Hans D.: Dagstuhl-Manifest zur Strategischen Bedeutung des Software Engineering in Deutschland. In: *Perspectives Workshop.* Dagstuhl, Germany : Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006 (Dagstuhl Seminar Proceedings 05402). – ISSN 1862–4405

[48] Chakraborty, Kanad: Testing and Reliability Techniques for High-Bandwidth Embedded RAMs. In: *J. Electron. Test.* 20 (2004), February, 89–108. `http://dx.doi.org/10.1023/B:JETT.0000009316.94309.66`. – DOI 10.1023/B:JETT.0000009316.94309.66. – ISSN 0923–8174

[49] Chang, M.-F. ; Fuchs, W. K. ; Patel, J. H.: Diagnosis and Repair of Memory with Coupling Faults. In: *IEEE Trans. Comput.* 38 (1989), April, 493–500. `http://dx.doi.org/10.1109/12.21142`. – DOI 10.1109/12.21142. – ISSN 0018–9340

[50] Chen, C. L.: Error-correcting codes for semiconductor memories. In: *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture.* New York, NY, USA : ACM, 1984. – ISBN 0–8186–0538–3, S. 245–247

[51] Chen, Chin-Long: *Symbol level error correction codes which protect against memory chip and bus line failures.* United States Patent 7093183. `http://www.freepatentsonline.com/7093183.html`. Version: 02 2001

[52] Cook, Jia-Rui C.: *Engineers Diagnosing Voyager 2 Data System.* `http://www.jpl.nasa.gov/news/news.cfm?release=2010-151`. Version: 05 2010, Abruf: 2010-05-20

[53] Corbet, Jonathan ; Rubini, Alessandro ; Kroah-Hartman, Greg: *Linux Device Drivers, 3rd Edition.* O'Reilly, 2005

[54] Corporation, Hewlett-Packard ; Corporation, Intel ; Corporation, Microsoft ; Ltd., Phoenix T. ; Corporation, Toshiba: Advanced Configuration and Power Interface Specification 4.0 - Chapter 14, System

Address Map Interfaces / Advanced Configuration & Power Interface. 2009. – Forschungsbericht

[55] Dekker, R. ; Beenker, F. ; Thijssen, L.: Fault modeling and test algorithm development for static random access memories. In: *Test Conference, 1988. Proceedings. New Frontiers in Testing, International*, 1988. – ISSN 1089–3539, S. 343 –352

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

[57] Dietze, Rolf ; Heuser, Tatjana ; Schilling, Jörg: *OpenSolaris für Anwender, Administratoren und Rechenzentren OpenSolaris für Anwender, Administratoren und Rechenzentren*. Springer, Berlin, 2006

[58] Dijkstra, E. W.: On the role of scientific thought. 1974. – Reprinted in Dijkstra's *Selected Writings on Computing: A Personal Perspective*, 1982, pp. 60–66.

[59] Eidus, Izik: *Linux Kernel Documentation: How to use the Kernel Samepage Merging feature*. [Kernel Source 2.6.34RC4]/Documentation/vm/ksm.txt, 11 2009

[60] Fengguang, Wu: *[PATCH 00/22] HWPOISON: Intro (v5)*. 06 2009

[61] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns*. Boston, MA : Addison-Wesley, 1995 `http://www.amazon.co.uk/exec/obidos/ASIN/0201633612/citeulike-21`. – ISBN 0201633612

[62] Gorman, Mel: *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004

[63] Haasl, D. F.: Advanced Concepts in Fault Tree Analysis. In: *Proceedings of the System Safety Symposium* The Boeing Company, 1965

[64] Hamming, Richard W.: Error Detection and Error Correction Codes. In: *The Bell System Technical Journal* XXVI (1950), Nr. 2, S. 147–160

[65] Hayes, J. P.: Detection of Pattern-Sensitive Faults in Random-Access Memories. In: *IEEE Trans. Comput.* 24 (1975), Nr. 2, S. 150–157. http://dx.doi.org/10.1109/T-C.1975.224182. – DOI 10.1109/T–C.1975.224182. – ISSN 0018–9340

[66] Henkel, Jörg: *Priority Programme 1500 "Design and Architectures of Dependable Embedded Systems - A Grand Challenge in the Nano Age"*. 2009

[67] Higginson, Peter L. ; Berent, Anthony N.: *Extended ECC system.* United States Patent 6425106. http://www.freepatentsonline.com/6425106.html. Version: 09 1999

[68] Hoare, C. A. R.: Algorithm 64: Quicksort. In: *Commun. ACM* 4 (1961), Nr. 7, S. 321. http://dx.doi.org/10.1145/366622.366644. – DOI 10.1145/366622.366644. – ISSN 0001–0782

[69] Huang, Rei-Fu ; Lai, Yan-Ting ; Chou, Yung-Fa ; Wu, Cheng-Wen: SRAM delay fault modeling and test algorithm development. In: *Proceedings of the 2004 Asia and South Pacific Design Automation Conference.* Piscataway, NJ, USA : IEEE Press, 2004 (ASP-DAC '04). – ISBN 0–7803–8175–0, 104–109

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

[71] Inc., MoSys: *Mosys adds soft-error protection, correction.* 2002

[72] Intel: *Nehalem whitepaper.* http://www.intel.com/technology/architecture-silicon/next-gen/319724.pdf,

[73] Intel: *Intel E7500 Chipset MCH Intelx4 Single Device Data Correction (x4 SDDC) Implementation and Validation.* Application Note (AP-726), August 2002

[74] International Organization for Standardization (Hrsg.): *ISO/IEC 9899-1999: Programming Languages—C.* http://www.ansi.org/: International Organization for Standardization, Dezember 1999

[75] Norm ISO/IEC FDIS 9126-1 2000. *International Organization for Standardization: Information technology - Software product quality. Teil 1: Quality model*

[76] KARPOVSKY, M. G. ; GOOR, A. J. d. ; YARMOLIK, V. N.: Pseudo-exhaustive word-oriented DRAM testing. In: *Proceedings of the 1995 European conference on Design and Test.* Washington, DC, USA : IEEE Computer Society, 1995 (EDTC '95). – ISBN 0–8186–7039–8, 126–

[77] KLEEN, Andi: *Linux Kernel Documentation: Layout for x86_64*. [Kernel Source 2.6.34RC4]/Documentation/x86/x86_64/mm.txt, 07 2004

[78] KLEEN, Andi: *Patch: Add 4GB DMA32 zone*. http://lwn.net/Articles/152337/. Version: 11 2005, Abruf: 2010-04-01

[79] KLEEN, Andi ; WU, Fengguang: *HW Posion Implementation.* Kernel sources/mm/memory-failure.c,

[80] KNUTH, Donald E.: Structured Programming with goto Statements. In: *Computing Surveys* 6 (1974), December, Nr. 4, S. 261–301

[81] KROAH-HARTMAN, Greg: *Linux Kernel Documentation: The Linux Kernel Driver Interface.* [Kernel Source 2.6.34RC4]/Documentation/stable_api_nonsense.txt,

[82] KUZNETSOV, N. Y. ; MIKHALEVICH, K. V.: Reliability Analysis of Systems Described by Fault Trees with Efficiency. In: *Cybernetics and Sys. Anal.* 39 (2003), September, 746–752. http://dx.doi.org/10.1023/B:CASA.0000012095.99836.63. – DOI 10.1023/B:CASA.0000012095.99836.63. – ISSN 1060–0396

[83] LEVINE, L. ; MEYERS, W.: Special Feature: Semiconductor Memory Reliability with Error Detecting and Correcting Codes. In: *Computer* 9 (1976), October, 43–50. http://dx.doi.org/10.1109/C-M.1976.218410. – DOI 10.1109/C–M.1976.218410. – ISSN 0018–9162

[84] Li, J. ; Wu, C.: Memory fault diagnosis by syndrome compression. In: *Proceedings of the conference on Design, automation and test in Europe*. Piscataway, NJ, USA : IEEE Press, 2001 (DATE '01). – ISBN 0–7695–0993–2, 97–101

[85] Lin, Shu ; Jr., Daniel J. C.: *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983. – TUB-HH 2413-469 3

[86] Loughmiller, Daniel: *Quality and Reliability of Semiconductor Devices*. `http://www.ee.uidaho.edu/ee/classes/ee481f01/Micronrelib.pdf`,

[87] Love, Robert: *Linux Kernel Development*. Pearson Education, Inc., 2005. – ISBN 3–446–22566–8

[88] MacWilliams, F.J. ; Sloane, N.J.A.: *The Theory of Error-Correcting Codes*. 2nd. North-holland Publishing Company, 1978

[89] Mauerer, Wolfgang: *Linux Kernelarchitektur*. Hanser, 2004. – ISBN 3–446–22566–8

[90] Meyer, Bertrand: *Building bug-free O-O software: An introduction to Design by Contract(TM)*. `http://www.eiffel.com/developers/design_by_contract_in_detail.html`, Abruf: August 2009

[91] Meyer, Bertrand: Applying "'Design by Contract"'. In: *IEEE COMPUTER* 25 (1992), S. 40–51

[92] Milojicic, Dejan ; Messer, Alan ; Shau, James ; Fu, Guangrui ; Munoz, Alberto: Increasing relevance of memory hardware errors: a case for recoverable programming models. In: *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*. New York, NY, USA : ACM, 2000. – ISBN 1–23456–789–0, S. 97–102

[93] Mukherjee, Shubhendu S. ; Emer, Joel ; Fossum, Tryggve ; Reinhardt, Steven K.: Cache scrubbing in microprocessors: Myth or necessity. In: *In 10th IEEE Pacific Rim International Symposium on Dependable Computing*, 2004, S. 37–42

[94] Mukherjee, Shubu: *Architecture Design for Soft Errors*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008. – ISBN 0123695295, 9780123695291

[95] Nair, R. ; Thatte, S.M. ; Abraham, J.A.: Efficient Algorithms for Testing Semiconductor Random-Access Memories. In: *Computers, IEEE Transactions on* C-27 (1978), june, Nr. 6, S. 572 –576. `http://dx.doi.org/10.1109/TC.1978.1675150`. – DOI 10.1109/TC.1978.1675150. – ISSN 0018–9340

[96] Needham, D. M. ; Jones, S. A.: A software fault tree key node metric. In: *J. Syst. Softw.* 80 (2007), September, 1530–1540. `http://dx.doi.org/10.1016/j.jss.2007.01.042`. – DOI 10.1016/j.jss.2007.01.042. – ISSN 0164–1212

[97] Neumann, John von: First Draft of a Report on the EDVAC. In: *IEEE Ann. Hist. Comput.* 15 (1993), Nr. 4, S. 27–75. `http://dx.doi.org/10.1109/85.238389`. – DOI 10.1109/85.238389. – ISSN 1058–6180

[98] Normand, E.: Single event upset at ground level. In: *Nuclear Science, IEEE Transactions on* 43 (1996), Nr. 6, 2742–2750. `http://dx.doi.org/10.1109/23.556861`. – DOI 10.1109/23.556861

[99] Olarig, Sompong P.: *Technique for implementing chipkill in a memory system*. United States Patent 7096407. `http://www.freepatentsonline.com/7096407.html`. Version: 02 2003

[100] Potyra, S. ; Sieh, V. ; Cin, M. D.: 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. – ISBN 978–1–59593–725–4, S. 9

[101] Ramamoorthy, C. V. ; Ho, G. S. ; Han, Y. W.: Fault tree analysis of computer systems. In: *AFIPS '77: Proceedings of the June 13-16, 1977, national computer conference*. New York, NY, USA : ACM, 1977, S. 13–17

[102] Riedel, M. ; Rajski, J.: Fault coverage analysis of RAM test algorithms. In: *Proceedings of the 13th IEEE VLSI Test Symposium*. Washington, DC, USA : IEEE Computer Society, 1995 (VTS '95). – ISBN 0–8186–7000–2, 227–

[103] Rodriguez, Claudia S. ; Fischer, Gordon ; Smolski, Steven: *The Linux Kernel Primer*. Prentice Hall, 2006. – ISBN 0–13–118163–7

[104] Romack, Rob: *Service Management Facility (SMF) in the Solaris 10 Operating System*. Edition: February 2006. Part No 819-5150-10: Sun, 02 2006

[105] Schroeder, Bianca ; Pinheiro, Eduardo ; Weber, Wolf-Dietrich: DRAM Errors in the Wild: A Large-Scale Field Study. In: *SIGMETRICS/Performance'09*, 2009

[106] Semiconductors, International Technology R.: *Executive Summary 2009*. http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_ExecSum.pdf, 2009

[107] Shapiro, Michael W.: Self-Healing in Modern Operating Systems. In: *Queue* 2 (2005), Nr. 9, S. 66–75. `http://dx.doi.org/http://doi.acm.org/10.1145/1039511.1039537`. – DOI http://doi.acm.org/10.1145/1039511.1039537. – ISSN 1542–7730

[108] Silberschatz, Abraham ; Galvin, Peter B. ; Gagne, Greg: *Applied operating system concepts*. 1. ed. New York [u.a.] : Wiley, 2000. – ISBN 0–471–36508–4

[109] Singh, Amandeep ; Bose, Debashish ; Darisala, Sandeep: Software Based In-System Memory Test for Highly Available Systems / SUN. 2005 (http://dx.doi.org/10.1109/MTDT.2005.34). – Forschungsbericht

[110] Sivathanu, Gopalan ; Wright, Charles P. ; Zadok, Erez: Ensuring data integrity in storage: techniques and applications. In: *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and*

survivability. New York, NY, USA : ACM, 2005. – ISBN 1–59593–233–X, S. 26–36

[111] Srini, V. P.: Fault Location in a Semiconductor Random-Access Memory Unit. In: *IEEE Trans. Comput.* 27 (1978), April, 349–358. http://dx.doi.org/10.1109/TC.1978.1675107. – DOI 10.1109/TC.1978.1675107. – ISSN 0018–9340

[112] Sun: *OpenSPARC T2 System-On-Chip (SOC) Microarchitecture Specification.* 05 2008

[113] Sun and Intel (Hrsg.): *The Solaris OS and Intel Nehalem-EX.* http://sun.com/solaris/intel: Sun and Intel, 09 2009

[114] Sun Microsystems, Inc. (Hrsg.): *Eversholt Fault Tree Description Language.* Part No: Eversholt 1.8: Sun Microsystems, Inc., 11 2008

[115] Takeuchi, K. ; Shimohigashi, K. ; Kozuka, H. ; Toyabe, T. ; Itoh, K. ; Kurosawa, H.: Origin and characteristics of alpha-particle-induced permanent junction leakage. In: *IEEE Transactions on Electron Devices* 37 (1990), März, S. 730–736. http://dx.doi.org/10.1109/16.47779. – DOI 10.1109/16.47779

[116] Talluri, Madhusudhan ; Hill, Mark D.: Surpassing the TLB performance of superpages with less operating system support. In: *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems.* New York, NY, USA : ACM, 1994. – ISBN 0–89791–660–3, S. 171–182

[117] Thompson, Doug: *Linux Kernel Documentation: EDAC - Error Detection And Correction.* [Kernel Source 2.6.34RC4]/Documentation/edac.txt, 06 2007

[118] Wu, Chi F. ; Huang, Chih T. ; Wang, Chih W. ; Cheng, Kuo L. ; Wu, Cheng W.: Error catch and analysis for semiconductor memories using march tests. In: *Proceedings of the 2000 IEEE/ACM international*

conference on Computer-aided design. Piscataway, NJ, USA : IEEE Press, 2000 (ICCAD '00). – ISBN 0–7803–6448–1, 468–471

[119] Yoon, Doe H. ; Erez, Mattan: Virtualized and flexible ECC for main memory. In: *SIGARCH Comput. Archit. News* 38 (2010), March, 397–408. `http://dx.doi.org/10.1145/1735970.1736064`. – DOI 10.1145/1735970.1736064. – ISSN 0163–5964