technische universität
dortmund

Masterthesis

Memory Access Analysis and Endurance Leveling Approaches for Non-volatile Working Memory Systems

Christian Hakert

July 25, 2019

Supervisors:
Prof. Dr. Jian-Jia Chen
Dr.-Ing. Kuan-Hsun Chen

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
http://ls12-www.cs.tu-dortmund.de

Abstract

Emerging technologies for non-volatile memory (NVM), such as phase-change memory (PCM), ferroelectric RAM (FeRAM), spin-transfer torque magnetoresistive RAM (STT-MRAM) and many more, have been considered as a replacement for DRAM and storage due to low leakage power, high capacity and fast access times. A major disadvantage of some NVMs is the significantly lower write endurance compared to DRAM. To tackle this issue, several in-memory wear-leveling approaches have been proposed in literature, targeting the program execution to make the memory write usage more uniform. These approaches operate on different granularities, like memory pages, cache-lines, big segments, etc. Aging-aware approaches take the current wear-level of memory cells into account to make the best wear-leveling decisions, while other approaches try to balance the wear-level in a random based manner or in a circular manner. However, most approaches propose specialized hardware controllers to collect the aging information and to perform wear-leveling actions, like relocations of memory regions.

As specialized hardware may be hard to build at all, this thesis proposes so-called software only wear-leveling, which only makes use of commonly available hardware components. Aging-aware wear-leveling on the granularity of virtual memory pages is achieved by estimating the cell aging by a statistical runtime approximation. As an extension to this, fine-grained wear-leveling for the `stack` region of applications is achieved by relocating the `stack` periodically in a circular manner through a reserved memory region.

The necessity for commonly available hardware components only allows the techniques to be executed in a full system simulation setup (i.e. gem5 as a simulator for an ARM CPU and NVMain 2.0 as a simulator for the NVM) for evaluation purposes. The results show that coarse-grained aging-aware wear-leveling with approximated cell ages works out, but cannot resolve non-uniform write distributions within virtual memory pages. This limits the gained improvement of the memory lifetime up to a factor of ≈ 13 compared to the baseline without any wear-leveling. Combining the coarse-grained approach with the fine-grained stack wear-leveling technique, the non-uniformity caused by the stack region can be resolved and the evaluations result in an improvement of the memory lifetime up to a factor of ≈ 900 compared to the baseline.

Acknowledgements

I would like to thank my supervisors Prof. Dr. Jian-Jia Chen and Dr.-Ing. Kuan-Hsun Chen from TU Dortmund for their great support. Productive discussions about new ideas and concepts were always welcome, which finally defined the scope of this thesis. Both of my supervisors encouraged me to bring in own ideas and practically try out my ideas at any time.
Furthermore i would like to thank the research team from the *Non-volatile Memory - One Memory Architecture* (NVM-OMA) project, supported by the german research group (DFG), to make this thesis part of a mature scientific project. They gave me the possibility to join an international workshop on the topic and to present my ideas there. Especially the research group of Prof. Dr.-Ing. Jörg Henkel from the Karlsruhe Institute of Technology (KIT) took part in the research for this thesis by refining and re-evaluating the concepts. The direct corporation with Paul Genßler from KIT helped me to make this thesis as complete as it is now.
I would also like to thank the Wilo-Foundation for supporting me with a scholarship in the scope of the Deutschlandstipendium during my studies.
Finally, i'd like to express my sincere gratitude to all my family and my friends for unfailingly supporting me in the last years. This thesis would be unthinkable without them.

Thank you.

Contents

1 Introduction **1**
1.1 Non-Volatile Memory Technologies 2
1.1.1 Phase-Change Memory (PCM) 3
1.1.2 Ferroelectric RAM (FeRAM) 4
1.1.3 Spin-Transfer Torque Magnetoresistive RAM (STT-MRAM) 4
1.2 Endurance Properties 5
1.2.1 Considerations About Memory Lifetime 6
1.3 Memory Write Access Patterns 7

2 Related Work **10**
2.1 Coarse-Grained, Aging-Aware Leveling 10
2.1.1 Random Based Wear-leveling 10
2.1.2 Aging-Aware Wear-leveling 11
2.2 Fine-grained Leveling in the Stack Region of C++ Applications 13
2.2.1 General Purpose Fine-grained Wear-leveling 13
2.2.2 Dedicated Heap and Stack Wear-leveling 14

3 Memory Access Analysis **16**
3.1 Simulation Environment 18
3.1.1 Configuration of gem5 and NVMain2.0 18
3.1.2 Bare-Metal Runtime System 20
3.1.3 Performance Counter Extension for gem5 28
3.2 Application Separation 29
3.2.1 Spatial Separation 30
3.2.2 Interrupt Isolation 31
3.3 Program Region Analysis 32

4 Page Based Memory Relocation **35**
4.1 Software Only Endurance Tracking 36
4.1.1 Write Distribution Recording 37
4.1.2 Abstraction of Physical Properties 40
4.1.3 Granularity and Overhead Considerations 41

4.2 Online Endurance Balancing Algorithm . . . 42
4.2.1 RBTree Based Page Management . . . 42
4.2.2 Integration in the Tracking System . . . 44
4.2.3 Relocation Implementation . . . 45
4.3 Evaluation . . . 46
4.3.1 Endurance Tracking . . . 46
4.3.2 Endurance Leveling . . . 48
4.3.3 Overhead . . . 52
4.3.4 Comparison to Start Gap Wear-leveling [26] . . . 54
4.4 Discussion . . . 56
4.4.1 Summary of Page Based Memory Relocation . . . 57
4.4.2 Summary of Hardware Requirements . . . 57
4.4.3 Considerations About Memory Lifetime . . . 58
4.4.4 Consideratons for Using a Software Only Approach . . . 59

5 Stack Region Write Access Leveling 60

5.1 Stack Frame Relocation . . . 63
5.1.1 Circular Stack Movement . . . 63
5.1.2 Synchronous Relocation Implementation . . . 66
5.1.3 Interrupt Relocation Implementation . . . 67
5.2 Address Consistency . . . 69
5.2.1 Raw Pointer Adjustment . . . 70
5.2.2 Smart Pointer Adjustment . . . 72
5.3 Synchronous Hinting Environment . . . 75
5.3.1 Hinting Decision . . . 76
5.3.2 Self Adaptive Loop Hinting . . . 77
5.3.3 Self Adaptive Recursion Hinting . . . 78
5.4 Combination With Page Based Relocation . . . 78
5.5 Evaluation . . . 80
5.5.1 Benchmark Setup . . . 80
5.5.2 Stack Only Leveling . . . 81
5.5.3 Combined Leveling . . . 87
5.5.4 Overhead . . . 92
5.6 Discussion . . . 94

6 Conclusion 96

6.1 Summary of Techniques . . . 96
6.2 Summary of Insights . . . 97
6.3 Comparison to Related Work . . . 98

6.4 Future Outlook 99

List of Figures **102**

List of Source Codes **103**

Bibliography **107**

Eidesstattliche Versicherung **108**

1 Introduction

Emerging technologies for non-volatile memory (NVM) have been considered to be used as a replacement for DRAM recently. Since these technologies are usually byte-addressable and have read and write latencies comparable to DRAM [13], this consideration is reasonable. A major difference between DRAM and most NVM technologies, like phase-change memory (PCM) or resistive RAM (ReRAM), is the significantly lower write endurance. While DRAM usually endures more than 10^{15} writes per cell, PCM endures $10^8 - 10^9$ writes per cell [13]. For instance, if a program wears out a DRAM cell within a time period of 10 years, it has to write the same memory cell every 900th CPU cycle on a 3GHz CPU. In comparison, the same write behavior would wear out a PCM cell within 5 minutes. According to this, when NVM technologies with low write endurance act as the main memory of a system, the system has to care about not to write the same memory cells too frequent, to not wear out the memory after a short time period.

Analyzing the memory write distribution of typical applications, a high non-uniformity in the write count per memory cell can be observed [26]. Most of the memory writes target only a few bytes, resulting in small memory regions with high write counts compared to the rest of the memory. Accordingly, when running typical systems and applications trivially on a NVM as main memory, the memory might wear out and fail early, making the system unusable. To tackle this problem, different wear-leveling or endurance leveling approaches have been proposed [18, 5, 16, 30, 22, 15, 21, 23]. These approaches take different information into account and usually work on different granularities. The majority of the proposed approaches is aging-aware, which means that the total write count to each memory cell is taken into account for the wear-leveling decisions. Usually, the write counts are assumed to be provided by the memory hardware.

In this thesis, the concepts of wear-leveling for main memory (i.e. aging-aware wear-leveling [18, 5, 16, 30] and wear-leveling for special program regions like the `stack` [22, 15, 21, 23]) are re-evaluated. The allover scope is to provide reasonable modifications to existing wear-leveling approaches, respectively provide alternative approaches, to enable aging-aware wear-leveling without any special support from the memory hardware, such as a write count per cell. After giving an introduction about the basic aspects of NVM technologies and their important properties regarding wear-leveling, the state-of-the-art and related work in literature is discussed. The subsequent contributions are arranged in three main topics:

- In Chapter 3, a setup to analyze the write behavior of typical applications is provided. Experimental evaluations of benchmark applications in this setup show how the memory writes are distributed. Concluding from the experimental results, necessary wear-leveling actions are discussed.
- Inspired by the proposed wear-leveling schemes in literature, an aging-aware wear-leveling scheme on the granularity of virtual memory pages is provided in Chapter 4. Contrasting to the most existing aging-aware approaches, the need for special hardware support is avoided by estimating the write counts during runtime. The estimation is done by involving the support of CPU integrated performance counters and the memory management unit (MMU). The provided wear-leveling scheme is experimentally evaluated on benchmark applications and the results are discussed. The discussion points out the need for finer-grained wear-leveling in some cases.
- As a generic wear-leveling scheme on finer granularities than virtual memory pages usually requires special hardware support, a specific wear-leveling scheme for the `stack` region of C++ applications is discussed in Chapter 5. The stack frame of the running application is periodically moved around a memory region, which leads to small and heavy written memory regions inside of the stack being moved around the memory. This resolves the high non-uniformity of write counts within the virtual memory pages of the stack region. As a last experimental evaluation, this approach is analyzed in combination with the previously described approach for wear-leveling on virtual memory page granularity.

This thesis concludes with the overall discussion of the provided approaches for wear-leveling. Especially the insights, advantages and limitations compared to proposed approaches in the literature are discussed. In the end, a future outlook for possible modifications and extensions is given.
The rest of this chapter gives an overview about the basic details of the NVM technologies. First, different NVM technologies are explained and compared. Based on the technology details, the reasons for the lower write endurance are stated. In the end of this introduction chapter, the influence of code execution on the memory usage is discussed.

1.1 Non-Volatile Memory Technologies

A classic DRAM cell consists of a transistor and a capacitor [20]. The capacitor can be charged and discharged by the transistor. The electrical charge inside the capacitor determines the stored information of the DRAM cell. Due to leakage, the capacitor looses the charge over time. If the charge is not refreshed periodically, the DRAM cell losses the stored information, which makes the memory volatile.

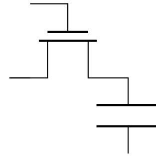

Figure 1.1: Architecture of a DRAM Cell [20]

Instead of changing electrical properties like the charge, most NVM technologies change physical properties of materials to store the information of the memory cells. As these changes persist over a longer time, the memory is relatively non-volatile. Memory cells are assembled into bigger memory arrays to build an entire memory device. The subsequent subsections give an overview of some existing NVM technologies and the physical properties saving the cell state.

1.1.1 Phase-Change Memory (PCM)

Phase-change memory stores the information by changing the electrical resistance of a phase-change material [13]. The electrical resistance can be measured and thus, the information can be read. The change of the resistance is achieved by bringing the material into either an amorphous or a crystalline state. The change of the state can be achieved by heating up the Phase-change material in different ways.

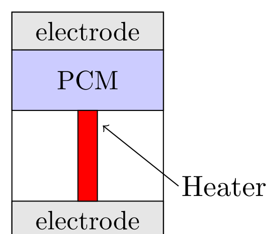

Figure 1.2: Architecture of a PCM Cell [13]

Figure 1.2 shows the architecture of a PCM cell. The Phase-change material is connected to electrodes to measure the electrical resistance. The heater is used to heat up the material and thus, program the cell. By heating up the material to a high temperature and letting it cool down quickly, the material reaches the amorphous state. In this state, the resistance is high and the logical state of the cell is 0. To program the cell with a logic 1, the material is heat up to a lower temperature and is kept at this temperature over a longer time. After cooling down, the material reaches a crystalline state with a low electrical resistance. The difference in the resistance of the amorphous and crystalline state is big enough to even store multiple levels in one PCM cell [13].

1.1.2 Ferroelectric RAM (FeRAM)

Ferroelectric RAM is one of the most mature NVM technologies by now. It is already mass produced and used in many products and applications [13]. The technology of FeRAM is very similar to the DRAM technology. While DRAM stores an electric charge inside a capacitor, FeRAM replaces the capacitor with a ferroelectric capacitor [24]. The ferroelectric element inside the capacitor can be polarized by applying an electric field to the element.

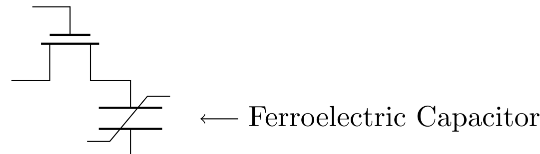

Figure 1.3: Architecture of a FeRAM Cell [24]

The polarization of the ferroelectric element determines the state in the cell. The cell is read by measuring the charge transferred through the capacitor and comparing it to a reference cell [24]. The polarization of the ferroelectric element influences the transferred charge so that at least two states (logic 0 and 1) can clearly be distinguished. In contrast to DRAM, the polarization of the ferroelectric element does not fade away due to leakage. This means, after powering off the device, the state of the cell keeps stable, thus FeRAM is non-volatile.

1.1.3 Spin-Transfer Torque Magnetoresistive RAM (STT-MRAM)

The NVM technologies presented above typically stress the used materials very much, which leads to low endurance of these technologies. Magnetoresistive RAM (MRAM) in general changes the magnetic orientation of a storage material [11]. Due to the effect of magnetoresistance the electrical resistance also changes by changing the magnetic orientation. Thus, different magnetic orientations represent different logical states of the cell, which can be measured by the electrical resistance. The technology of STT-MRAM has been reviewed in detail by Bhatti et al. in [11].
To change the magnetic orientation of a storage material, MRAM technologies applied magnetic fields to the material in the past. As this is complex to realize, STT-MRAM applies a simpler method. By applying a spin-polarized current, the magnetic orientation of a cell is changed. Normally, half of the electrons of a current flow have up-spin and the other half has down-spin. By passing a current through a fixed magnetic layer, called pinned layer, the electrons with parallel spin to the magnetic orientation of the pinned layer pass the layer, the other electrons are scattered back. The resulting polarized current is applied to the storage layer, called free layer. The magnetic orientation of the free layer

is changed parallel to the spin of the polarized current by this process. This process is illustrated in Figure 1.4.

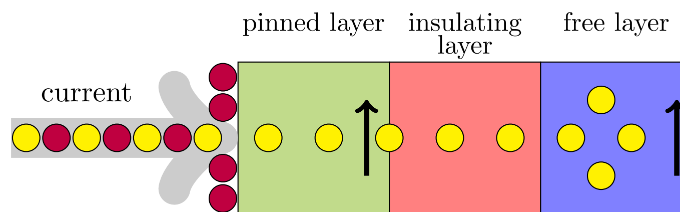

Figure 1.4: Programming a STT-MRAM Cell
The black arrows show the magnetic orientation of the pinned and free layer. The yellow electrons have a spin, which is parallel to the pinned layer, the purple electrons have a spin, which is antiparallel to the pinned layer. Because only the yellow electrons pass the pinned layer, the polarized current goes to the free layer and influences the magnetic orientation according to the spin of the electrons of the polarized current.

As the pinned layer has a fixed magnetic orientation, the cell cannot be reset in the same way. To reset the cell, the current flow is inverted. The unpolarized current then goes through the free layer and is filtered in the pinned layer. Electrons with an anitparallel spin to the pinned layer are scattered back and remain in the free layer, while electrons with a parallel spin to the pinned layer pass the pinned layer. Due to the concentration of the electrons with an antiparalllel spin in the free layer, the magnetic orientation of the free layer changes to the antiparallel orientation of the pinned layer. Figure 1.5 illustrates this process.

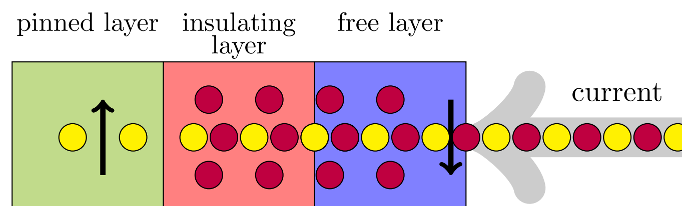

Figure 1.5: Resetting a STT-MRAM Cell

As mentioned before, the magnetic orientation of the free layer influences the electrical resistance of the free layer, due to the effect of magnetoresistance. The electrical resistance can be measured and the cell can be read. The magnetic orientation again is a property, which remains after powering off the device, which makes STT-MRAM non-volatile.

1.2 Endurance Properties

Section 1.1 presents some NVM technologies. All in all, it turns out that the technologies use different physical properties to store the memory values. Due to the different prop-

erties, different procedures are required to read and write values into the memory cells. This leads to different characteristics of the memory types. The procedures for reading and writing influence the read and write latency of the memory device. The overall architecture of the memory cells determines the endurance for each technology. Table 1.1 gives an overview of the write endurance and the read / write latency for the presented NVM technologies.

	DRAM	NAND Flash	PCM	FeRAM	STT-MRAM
Write endurance	$> 10^{15}$	$10^4 - 10^5$	$10^8 - 10^9$	$10^{14} - 10^{15}$	$10^{12} - 10^{15}$
Read latency	$\approx 10ns$	$15 - 35\mu s$	$20 - 60ns$	$20 - 80ns$	$2 - 35ns$
Write latency	$\approx 10ns$	$200 - 500\mu s$	$20 - 150ns$	$50 - 75ns$	$3 - 50ns$

Table 1.1: Overview of Memory Characteristics [13]

As the table shows, PCM has a significantly lower endurance than DRAM. The read and write latencies are comparable fast to DRAM, which means PCM might be a candidate to replace the DRAM as main memory. Even if some NVM technologies face better endurance than others, there are many other advantages and disadvantages, which may decide the used memory type in the end. Because of this, system software should be prepared to run on low endurance memory systems.

1.2.1 Considerations About Memory Lifetime

As stated above, some reasonable NVM technologies, like PCM or STT-MRAM, show a significantly lower write endurance than DRAM. When these memory types are used as main memory, the lifetime of the system is bound to the memory endurance. The allover assumption is, as soon as one memory cell is dead and cannot be programmed with a new value, the system is considered dead and cannot be used anymore. Thus, the most written memory cell in the system determines the allover lifetime.

Ideally, the system software would distribute the writes to the memory cells in a way, that all cells are written equally often[1]. Under this write pattern, the system would reach the maximal lifetime, as the first cell fails together with all other cells. Assuming the system software needs a fixed amount of memory writes to execute and the writes can be rearranged arbitrarily, the ideal write count for each cell would be the mean write count of all written cells under an arbitrary, unbalanced execution of the system software.

$$\text{mean_write_count} = \frac{\sum\limits_{\text{memory_cells}} \text{write_count_of_cell}}{\text{number_of_memory_cells}} \tag{1.1}$$

[1] For some memory types not only the write count determines the endurance, but generally the write count is a major factor.

Equation (1.1) shows how the ideal write count can be calculated, given the write counts for each memory cell under an arbitrary software run. The lifetime, which would result under the ideal write distribution, can be calculated according to Equation (1.2).

$$\text{ideal_lifetime} = \frac{\text{write_endurance}}{\text{mean_write_count}} \tag{1.2}$$

As explained before, in a real, unbalanced execution, the most written cell determines the allover lifetime, due to the fact that the memory is unusable as soon as the first cell is dead. According to this, the achieved lifetime for a concrete execution can be calculated according to Equation (1.3).

$$\text{achieved_lifetime} = \frac{\text{write_endurance}}{\text{max_write_count}} \tag{1.3}$$

To develop generalized solutions, not only the concrete lifetime is important, but also the improvement of lifetime is important. As no balancing scheme can achieve a better result than the ideal lifetime, different balancing schemes can be evaluated and compared regarding their achieved lifetime compared to the ideal lifetime.

$$\text{Achieved Endurance (AE)} = \frac{\text{achieved_lifetime}}{\text{ideal_lifetime}} = \frac{\text{mean_write_count}}{\text{max_write_count}} \tag{1.4}$$

Equation (1.4) shows the calculation of the achieved endurance (AE) value. This number shows, which fraction of the ideal lifetime was reached by a concrete software execution, no matter what the concrete endurance of the used memory was. The Achieved Endurance will later be used as a metric to evaluate the quality of different balancing schemes, but it can also be used to predict the expected memory lifetime, knowing the write endurance and the required execution time.

1.3 Memory Write Access Patterns

When NVM serves as the main memory of a system, the memory behavior of the running application determines the write pattern to the memory. Usually, a compiled program consists of different memory regions, which are used for different purposes.

- The `text` region holds the compiled machine code of the software. The CPU usually loads the contents out of this region and executes the instructions. During normal execution, the `text` region is not modified and thus, does not cause any write to the main memory.
- The `data` region contains all global variables and instances of the program. As the variables and members of the instances are changed during the program execution, the `data` region is written. The way, the region is written highly depends on the

program logic. The write pattern to this section might only be modified by changing the placement of the variables within the `data` region.

- The `bss` region (block storage segment) is very similar as the `data` region. The difference is, that the `bss` region only contains uninitialized data, which can be stored compressed in the compiled binary file. The write pattern to the `bss` region again strongly depends on the program logic and only might be changed by the placement of variables within the `bss` region.

- The `stack` region holds all local variables and additional information for the currently executed function. Each function call places a new frame on the top of the `stack` to store local variables and additional information, which is removed when the function call returns. Due to this, nested function calls let the used `stack` memory grow. The resulting write pattern highly depends on the program logic. The more local variables are used, the bigger the frame for the function calls is. For a lot of nested function calls, e.g. for recursive implementations, the writes to the `stack` region are distributed over a larger memory region, while for repeated, not nested function calls the frames for the function mostly reside at the same memory region and thus, the memory writes target a small memory region.

- The `heap` region does not belong to the compiled program directly. The `heap` is a software managed memory region to allocate global variables during runtime. Thus, the write pattern to the `heap` completely depends on the number of invocated `heap` allocations and deallocations and the management strategy for the heap.

In essence, the allover memory write pattern of an executed program consists of the different write patterns to the memory regions. The regions reside at different memory locations and thus, different memory locations are targeted by different write patterns from the program.

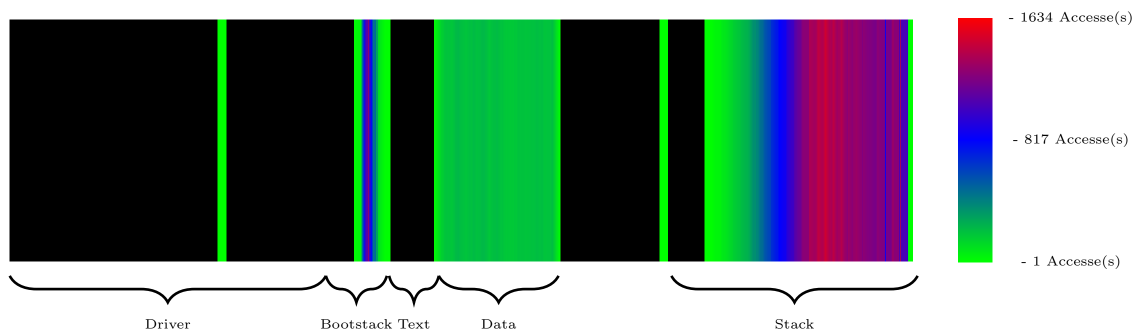

Figure 1.6: Memory Write Distribution to Memory Regions

Figure 1.6 shows an example visualization of the write distribution for an example program execution. The placement of the memory regions is indicated by the bottom labels. It can be seen, that each memory region faces a different write pattern, due to the different usage of the memory regions by the program. Furthermore, a high non-uniformity of write accesses within the entire memory can be observed, which would lead to a significantly lower memory lifetime. To resolve this non-uniformity, this thesis first proposes a coarse-grained approach to redirect writes to currently less written memory regions. As a high non-uniformity can be observed in the `stack` region at most, this thesis proposes an additional fine-grained approach to balance memory writes within the `stack` region.

2 Related Work

In the last years, several approaches for in-memory wear-leveling in NVM driven systems have been proposed in the literature. Some of these approaches are general purpose, whilst others only target specific NVM technologies (e.g. PCM). The design of the proposed approaches varies in different aspects. Some approaches are completely hardware based, some are completely software based and some propose hybrid solutions. Furthermore, one group takes runtime information (i.e. the write count or the wear-level of each cell) into account, while another group balances out the wear-level of the cells without using this information. However, no matter which decisions are made by the different approaches, the basic concept is always to change the physical memory locations of the memory, used by the application. This leads to writes being distributed better over the available memory and to a longer memory lifetime. As in this work two major concepts are presented, this chapter presents the regarding related work for both concepts.

2.1 Coarse-Grained, Aging-Aware Leveling

Several approaches aim to replace the location of sets of memory bytes from time to time. The size of these sets determines the granularity of the approach. The sets can be small (e.g. a cache-line [26, 27]), on the size of pages [14, 17, 5, 23] or even segments, consisting of multiple pages [30]. While the replacement of virtual memory pages or sets of virtual memory pages can be done in software, using the MMU, the management of smaller memory sets requires dedicated hardware support. This subsection first describes the coarse-grained approaches, which do not require runtime information about the memory usage. After this, the aging-aware approaches are presented and compared.

2.1.1 Random Based Wear-leveling

Qureshi et al. describe a scheme, which manages the used memory in cache-line sized blocks [26]. A mapping of logical cache-lines to physical memory regions is maintained and updated periodically. An additional spare cache-line, called gap, is moved through the physical memory in backward order. Each movement of the gap to the predecessor, causes the predecessor to take the former physical location of the gap. Thus, all cache-lines are moved through the memory in forward order. Once the gap passed the entire memory space, all cache-lines have been moved one line further. Repeating this process causes the

used memory space to be shifted around the physical memory in a circular manner. Doing this, memory writes to heavy written and less written cache-lines are equally distributed all over the physical memory. The required management information for this scheme is very small, because the mapping of the cache-lines can be calculated in hardware, by only knowing the current offset of the first cache-line and the position of the gap. The overhead for copying the cache-lines through the physical memory can be controlled precisely by the movement rate of the gap.

Ferreira et al. present a different approach, based on on a dedicated selection policy to swap memory regions, called pages, with the help of a logical to physical address mapping per page [17]. Basically, a mapping table for the physical mapping of each page is maintained. Whenever a page L is written back to the main memory and a "swap condition" occurs (e.g. the overflow of a global or per page write counter), a second target page P is selected and both pages are exchanged by exchanging the content and the physical mapping. The target page P may be selected as the least frequent written page, but due to the high complexity the target page is selected randomly. Theoretically, the presented approach is capable of arbitrary small page sizes, but the required mapping table size has to be taken into account. In the provided evaluation the size of pages is set to 2kB.

2.1.2 Aging-Aware Wear-leveling

Zhou et al. propose a coarse-grained, aging-aware segment swapping as a necessary extension, when a fine-grained wear-leveling already is applied [30]. They present a fine-grained approach, called row shifting, which evenly balances the memory writes to 1kB sized rows. As this happens very locally, there are still memory regions, which are used more excessive than others. The segment swapping mechanism solves this by managing the memory in coarse-grained segments (in the evaluation 1MB per segment) and exchanging the physical location of segments with the help of a hardware provided translation mechanism. For each segment, two control variables are maintained, the write count and a last-swapped timestamp. The write count is used to select a less written segment, called cold segment, to relocate an often written segment to this location. The last swapped timestamp is used to prevent a cold segment from being selected too often. The swapping of an often written segment is triggered when a threshold is exceeded by the write count of this segment. As the selection of the swapped segments and the selection of the target segments are made according to the write count, this approach is aging-aware. Zhou et al. propose to perform the segment swapping in the memory controller, thus this is a hardware based solution.

In contrast to the hardware based solution, Chen et al. propose a hybrid architecture, where the wear-leveling is performed by the operating system with the help of the MMU, while the necessary write counters for aging-aware wear-leveling are provided by the memory hardware [14]. The write count for each physical memory page is maintained by the

memory hardware and provided to the operating system. The operating system then uses this information to relocate often written pages or to make an aging-aware allocation decision for new pages. The relocation of pages is performed by modification of the pagetable, interpreted by the MMU. Chen et al. describe two algorithms to perform the wear-leveling decisions, bucket based wear-leveling and array based wear-leveling. For bucket based wear-leveling, pages are organized in different buckets. Each bucket contains pages with a write count in a certain range. Once the write count exceeds the range, the page is moved to the bucket with the corresponding range. Once a page reached the bucket, designated for the oldest (most written) pages, it is relocated to a young (less often written) page. The young pages are allocated from the bucket with the lowest write counts. Once this bucket is empty, the next bucket becomes the youngest bucket. The array based wear-leveling in contrast is more simple and requires less management data. All pages are organized in an array. A so-called pivot pointer splits the array into two halfs, young and old pages. If a relocation is necessary for an often written page, a young target page is selected from the young half of the array. The target page is moved into the old half of the array, which causes an adjustment of the pivot pointer. Once the pivot pointer reaches the end of the array (i.e. the young half is empty), the pointer wraps around and the entire array is considered young. In this work, Chen et al. describe two management strategies, which face different overheads and wear-leveling qualities. However, both approaches depend on hardware provided write counts at least per page.

The previously described two works perform aging-aware wear-leveling periodically. The trigger for relocation decisions usually is a write counter exceeding a threshold value or a timer running out. As these mechanisms naturally cause a certain overhead regarding execution time, memory writes and energy consumption, works have been proposed to balance the wear-level proactively by hooking into the memory allocation process. Khouzani et al. propose a wear resistant page allocation and deallocation mechanism, which is invoked from the memory allocation procedures of the operating system [5]. Whenever an application requests a new page, an appropriate page with an appropriate write count is selected. The work assumes, that the write characteristics for different memory segments of the application are well known by the operating system, thus page requests for read only segments can be allocated to already heavy written pages. For write heavy segment requests, pages with low write counts are allocated. An additional wear out prevention procedure deallocates extremely hot pages, before the physical memory wears out. By applying this scheme, no relocations are triggered to enable the wear-leveling, besides relocations for the wear out protection. The entire wear-leveling is performed in the allocation and deallocation process of pages. Again, the write count has to be provided by the hardware.

Li et al. propose a similar memory allocator, called UWLalloc [22]. Whenever the application requests memory from the operating system, a corresponding less written memory

region is returned from the allocator. Under the assumption, that allocations and deallocations happen periodically, the wear-level can be balanced. UWLalloc does not take a hardware provided write counter into account to select a less written page, but uses a self maintained allocation counter. A frequent allocated region will be allocated less frequent in future, according to this. Thus, contrasting to the previous aging-aware mechanisms this mechanism does not require special hardware support. However, this mechanism requires the applications to perform allocations and deallocations periodically. Additionally, the self maintained allocation counter strategy only works out, if the application performs approximately the same amount of writes to each allocated region. Furthermore, no wear-leveling within the allocated memory regions can be achieved.

2.2 Fine-grained Leveling in the Stack Region of C++ Applications

The related work in Section 2.1 aims to balance the wear-level by exchanging the physical location of sets of memory bytes. As a result, the wear-leveling happens only on a certain granularity and the uneven memory usage within this granularity is not resolved. This usually leads to unsatisfying results if only these approaches are applied. To tackle this problem, different approaches have been proposed. On the one hand, a wear-leveling on extreme fine granularities (cache-line granularity [26] or even bit granularity [30]) can be achieved. These approaches are applicable on any executed application, without limitations. Thus, they are meant to tackle the problem with a general purpose solution. On the other hand, an additional wear-leveling, targeting the problem sections with specialized solutions can be applied. As the program heap is managed by the operating system anyway, some works propose to design an aging-aware allocator for heap memory to resolve uneven wear-levels in the heap memory [22, 28]. Additionally, works propose to extend the usage of the program stack to the heap, by allocating heap memory for new function calls [22, 21]. In this subsection, first the general purpose approaches for fine-grained wear-leveling are presented. After this, the approaches for dedicated heap and stack leveling are described and compared to the solution, presented in this thesis.

2.2.1 General Purpose Fine-grained Wear-leveling

Qureshi et al. extend their scheme of periodically moved cache-lines in a circular manner by an address space randomization [26]. They observe, that heavy written cache-lines often have a spatial correlation, which is not resolved by relocating the entire address space in a circular manner. To resolve this, they propose an additional, static randomization of the address space, to resolve the spatial correlation of heavy written cache-lines. The randomization is integrated into the wear-leveling process by a two stage address transla-

tion. When the CPU requests a logical address, the randomizer generates an intermediate address, which is not correlated with similar heavy written cache-lines. This intermediate address is then processed by the wear-leveling mechanism and generates the physical address, which is passed to the memory. To ensure the correctness, the randomizer has to map each intermediate address to exactly one logical address to avoid double mappings. Qureshi et al. propose randomizer functions, which can be calculated on demand and thus, do not require additional storage overhead. For certain types of memory hardware, wear-leveling on a finer granularity than cache-lines has no effect, because for each write to a cache-line, all the memory cells within the cache-line are stressed equally. Thus, the address randomization of cache-lines can achieve a reasonable good result.

Zhou et al. propose an even finer wear-leveling, which is again additionally used as an extension to a coarse-grained, memory region swapping mechanism [30]. The fine-grained wear-leveling, called row shifting, is used to balance the wear-level within rows (1kB sized) and thus, the coarse-grained wear-leveling, called segment swapping, handles evenly balanced memory regions. The row shifting is achieved by shifting the bits of a row by an offset during the translation of a requested address to a physical address. This is done by additional memory hardware and a configurable shifter register. The shift offset can be arbitrary small, even down to a single bit, however the evaluations show that extreme fine shift granularities cannot achieve better results than shifting offsets of multiple bytes. Again, the overhead and quality of the resulting wear-leveling can be controlled by the frequency of modifications to the shifter register.

Both presented approaches for fine-grained, general purpose wear-leveling are completely hardware based. They require few management information and few interaction with the operating system. However it is not obviously clear, if such complex hardware mechanisms can be integrated into memory hardware easily and fast enough, to not slow down the memory hardware too much. Because of this, software only solutions, achieving similar results should be taken into consideration, as they are an alternative when the special memory hardware is not provided.

2.2.2 Dedicated Heap and Stack Wear-leveling

Two approaches for heap wear-leveling, proposed by Li et al. [22] and Khouzani et al. [5] are presented in Section 2.1. These approaches aim to level not only coarse-grained blocks of memory, but also resolve the uneven write patterns of application within the heap region. Li et al. propose a similar allocation mechanism for the usage of the program stack [21], as it faces highly non-uniform write accesses to the memory. The presented allocation mechanism is invoked, whenever a new function is called. Instead of calling the function on top of the existing stack region, a new memory region is allocated for

the function call in a wear-level aware manner. The function is then called on this newly allocated stack. When the function returns, the caller function continues on it's own stack. Li et al. present an extended approach in another work, which combines the allocation of heap and stack memory regions in one allocator [22]. The calling of functions is handled in the same way, but to allocate the called function's new stack region, the common allocator for heap and stack memory is invoked. This allows the management strategy to mix the applied usage pattern to the physical memory and thus achieve a better balance of the wear-levels.

Summing up, there is related work, which applies general purpose approaches to achieve fine-grained wear-leveling and there is related work, which specifically targets the stack and the heap region of program execution to resolve the observed non-uniformity within this regions. As the general purpose approaches usually require advanced hardware support, they are hardly applicable on existing hardware. Nevertheless, the achieved result of these approaches can be compared to the developed approaches in this thesis. The specialized approaches for heap and stack wear-leveling depend on a certain collaboration of the running application. The application has to perform heap allocations and function calls at a certain frequency to enable these approaches to achieve good results. Furthermore, allocations of new called function's stack frames on a heap might lead to some internal fragmentation, when the memory footprint of a function changes depending on their input parameters. The achieved result by these approaches is less comparable with the stack wear-leveling approach, presented in this thesis, because they not only differ in a management strategy for the memory, but in the way they are integrated into the running software. Still, both approaches can be compared as they are entirely software based and can be applied as an extension to coarse-grained, memory region swapping techniques.

3 Memory Access Analysis

As already stated in Chapter 1, this thesis is about to develop and evaluate software only solutions for wear-leveling in the main memory of software execution. Two major approaches are presented and compared to the related work and to the baseline, which is the software execution without any active wear-leveling. However, to analyze and evaluate the effects and improvements, achieved by the wear-leveling approaches, detailed information about the software execution needs to be collected. Section 1.2.1 gives an overview, which information is required to calculate the achievement in endurance improvement for a concrete technique. At least, the mean write count to each memory cell and the maximal write count over all memory cells has to be determined to evaluate a technique. For certain memory architectures it is not required to record the write count to each cell, because groups of memory cells might be always written together. For all subsequent evaluations in this thesis, writes are assumed to always cause a write to an entire cach-line of 64B. Given a byte exact address $\mathcal{A}$, which is accessed by the CPU, all memory bytes (and all cells within these bytes) $\mathcal{T}$ according to Equation (3.1) are assumed to be accessed[1].

$$\mathcal{T} = \mathcal{A}\&(\texttt{0xFFFFFFFFFFFFFFC0}) + i, i \in \{0, ..., 63\} \tag{3.1}$$

Under this assumption, it is totally sufficient to record the write count for every cach-line, which significantly reduces the required runtime overhead and storage overhead.
Even if recording write counts only on a cach-line granularity saves some overhead, usually there is no hardware support to record this information on standard computer hardware. The only counting mechanism for write accesses is usually provided by a CPU global performance counter, which counts all write accesses to the main memory (e.g. the `BUS_ACCESS_ST` event in Cortex-A53 CPU's performance monitor unit [2]). This mechanism cannot provide sufficient information to perform any evaluation, because no relation between the write count and the location of the writes is available. To overcome this issue, different solutions might be considerable to trace memory accesses precisely:

- Use special hardware, which is capable of tracing the exact memory accesses per cell / byte / cach-line. This could be realized by using an field programmable gate array (FPGA), which is mapped into the logical address space of the CPU. The

[1] Accesses are always assumed to be within one cach-line. If a memory access targets data, which is distributed over a group of cach-lines, multiple accesses (one per cach-line) are triggered by the CPU.

write accesses from the CPU to this FPGA mapped region can be redirected to a real connected memory module and furthermore, accesses can be stored, counted and accumulated.

- Use standard hardware with enhanced debug capabilities (e.g. JTAG). Using this, an external debugger could analyze every memory access and perform the counting and accumulation of the memory accesses.
- Use a full system simulator, which performs a cycle accurate simulation of an entire CPU with all peripherals like memory, controllers, buses, etc. Such a simulator can be easily modified to trace memory accesses, count write accesses and accumulate them. Furthermore, such a simulator can easily run inside a Linux host operating system and thus, the trace results can be easily stored and processed.

The concept of using an FPGA to trace memory accesses is proposed by Bao et al. [9]. An FPGA board is connected to a DIMM slot of a test system, instead of a DIMM memory module. The FPGA board itself contains a DIMM slot, so the memory requests are directly routed to a memory module. The FPGA board sniffs the memory requests and aggregates them or sends them out via an Ethernet interface. This approach requires a complex hardware setup and a certain operating system support to synchronize the memory accesses with the sniffer board. A similar architecture could be achieved more easily with an FPGA and CPU on the same SoC. However, it is out of the scope of this thesis.

Alternatively, a cycle accurate full system simulator, called gem5, is developed by Blinkert et al. [12]. gem5 is capable of simulating different CPU architectures, like ALPHA, ARM, X86, MIPS and many more [4], different CPU models, like simple CPUs without pipelining or complex CPUs with pipelining and out of order execution, and different machines with different connected controllers. This allows very accurate simulations of a realistic hardware system without the need for complex hardware setups, at the cost of simulation time. Due to the modular structure of gem5, several extensions can be plugged into the simulation process and extend the simulated hardware. NVMain2.0 is a memory simulator for non-volatile main memory systems, which can be plugged in as an extension for gem5 [25]. NVMain2.0 is triggered on every memory access and applies a physical model of a non-volatile memory architecture, which for instance influences the read and write access latency. Furthermore, NVMain2.0 is capable of generating trace files, which contain a set of detailed information for every access to the main memory. For every access, following details are stored in the trace file:

1. Memory controller cycle timestamp
2. Read or write access

3. Access address
4. Old memory content
5. Memory content after access

The generated trace files can be easily processed to accumulate the write count per memory byte / cach-line and evaluate a wear-leveling approach. Due to the simplicity of this approach, it is used in this thesis to simulate all proposed wear-leveling schemes and evaluate their quality according to the generated memory trace.
This chapter first describes the setup of gem5 and NVMain2.0 in detail in Section 3.1. An extension to enable a certain performance counter event, provided by this thesis, is also discussed. The simulation setup is not used to run a full featured operating system like Linux, but to run a specialized bare-metal runtime system, which is also provided by this thesis. The subsequently presented runtime system enables all the architectural support, which is required by the wear-leveling approaches. Section 3.2 presents the techniques, applied to separate the memory trace of a target application from the memory trace of the runtime system, afterwards. The chapter concludes with the description of evaluation and analysis techniques, which are applied on the generated trace files to evaluate the running application in Section 3.3.

3.1 Simulation Environment

As already explained, in this thesis a combination of the gem5 simulator and the NVMain2.0 memory simulator for non-volatile main memory systems is used. This section first gives an overview about the configuration and usage of both simulators. After this, the bare-metal runtime system is presented, which is executed in the simulator to run arbitrary test applications. In the end, an extension applied to gem5, is presented, which enables a special performance counter event, which is not supported by gem5 by default.

3.1.1 Configuration of gem5 and NVMain2.0

The sourcecode from both, gem5 and NVMain2.0, can be obtained from the official websites. Both are mostly written in C++ and Python, while Python is mostly used for configuration purposes. The sourcetree of NVMain2.0 has to be included in the building process of gem5, which can be easily achieved by the build system of gem5. NVMain2.0 itself comes with a patch for gem5 to achieve compatibility. Unfortunately, the most recent release of gem5 is no longer compatible with NVMain2.0, because some interfaces were changed. One reason is the renaming of a function[2], which can be easily patched in NVMain2.0. Another reason is the usage of smart pointer classes instead of raw pointers

[2] Changed in commit `2f17062dd9a465943b57723f72f89ec66a0db664` in the official gem5 git repository

for the memory request interface, which has to be patched accordingly in NVMain2.0. Including this patches, NVMain2.0 compiles together with the most recent version of gem5 and can be executed. Subsequently, this setup is always used.

NVMain2.0 can be highly configured, regarding the physical model applied on the memory access behavior. Several latencies and physical properties (frequency, chip organization, energy consumption, ...) can be configured in a centralized configuration file. NVMain2.0 includes a set of predefined configuration files for different NVM technologies. As this thesis does not focus on the special properties of concrete NVM technologies, the detailed configuration is less important. NVMain2.0 includes a minimal memory configuration, called `printtrace.config`, which sets the minimal required properties and enables the generation of the previously described trace files. In this thesis, the `printtrace.config` configuration is used for all simulations and evaluations, because no more details about the NVM type are necessary.

The gem5 simulator can also be configured to a certain degree. One key property is, that gem5 supports a so-called Systemcall Emulation (SE) and a Full System (FS) mode [12]. While the Full System modes simulates an entire system and thus, the executed software has to include all drivers and runtime libraries, the Systemcall Emulation mode can be used to run Linux applications in the simulation environment. The systemcalls, which are usually handled by Linux, are handled by the simulation environment, thus the entire stdlib and other libraries can be used. The simulation environment for this thesis aims to simulate an entire system, including device drivers and runtime libraries, thus the Full System mode is used in this thesis. Furthermore, the Systemcall Emulation mode offers limited control over the system, because the application is executed with user level privileges. The application cannot modify the virtual memory configuration, interrupt routines and other operting system related tasks.

In addition to the Full System mode configuration, gem5 has to be configured to simulate a concrete CPU and machine. As already stated, gem5 is capable of simulating CPUs with different complex architectures [4]. For the subsequent simulations, the `O3CPU` is used. This CPU is an out of order CPU with five pipeline stages. The machine configuration defines the additionally connected hardware to the CPU. This includes the interrupt controller, timers, UART controller, usb, ethernet and many more. For this thesis, the ARMv8 architecture is used as the machine architecture [1], thus an ARMv8 machine configuration is used. The `VExpress_GEM5_V2` machine configuration is a full featured configuration, including the controllers which can be found on common ARMv8 based SoCs. The detailed configuration of this machine is done in a centralized file within gem5[3]. This file also includes the memory mapping of the different controllers, which is required for the running system to run device drivers.

[3]In the gem5 sourcetree: `src/dev/arm/RealView.py`

3.1.2 Bare-Metal Runtime System

The description in Section 3.1.1 shows the detailed configuration of the simulation setup. According to the simulated system, the executed system has to provide device drivers and some library functions. This is typically provided by the operating system, but in this thesis no standard operting system is used. Instead, a special bare-metal runtime system is used, which provides the minimal required support. The reason for this is, that the wear-leveling techniques later require modifications of the operating system, which would be extremely complex to achieve in Linux, for instance. From an architectural point of view, the bare-metal runtime system is a part of the simulation environment and not part of the application, because it is used to run different test applications and is always part of the simulation setup. Figure 3.1 illustrates the architecture of the entire simulation setup.

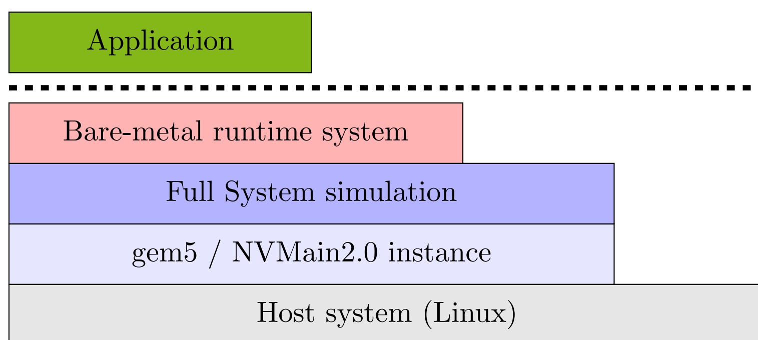

Figure 3.1: Architectural Overview of the Simulation Environment

This subsection presents details about the different components of the bare-metal runtime system. For each component, the purpose is discussed in detail. Basically, the gem5 setup simulates a realistic ARMv8 based chip and thus, the bare-metal runtime system can be modified to run on a real ARMv8 based chip easily by configuring the device drivers with the according memory mapping.

Bootcode

On a usual ARMv8 based CPU, several levels of bootloaders are executed, when the device is powered on. These bootloaders provide simple mechanism to load the target application (e.g. an operating system) from the storage / from the network to the memory and execute it. This is mostly done by loading a binary image to a fixed memory location and set the CPU's instruction pointer to this location afterwards. As gem5 runs on top of Linux, the loading of the target application can be done even simpler by passing the binary file as an argument to the gem5 command. Anyway, the binary file is loaded to a fixed memory location and the instruction pointer is set to this location. The bare-metal system has to make sure, the bootcode is placed at the very beginning of the binary

image, so the bootcode is the first thing executed. The placement of the bootcode is done by a custom linker configuration during the compilation of the bare-metal runtime system. The bootcode itself is written in assembly and prepares the CPU to execute C++ compiled code. This includes the setting of the interrupt table and the stack pointer to the according locations in the bare-metal system's memory. The stack pointer is set to a dedicated bootstack, because the applications require a separate stack later. Once this setup is done, the bootcode calls the entry function of the initialization code, which is written in C. Figure 3.2 gives an overview of the memory organization regarding the bootcode.

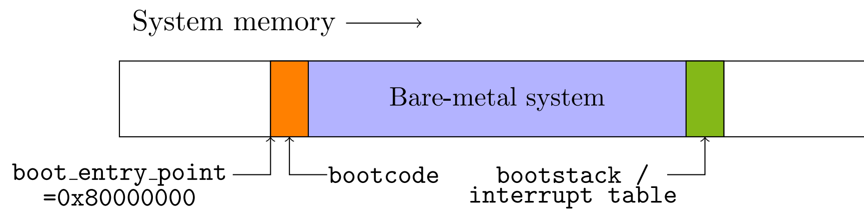

Figure 3.2: Memory Placement of the Bootcode

Initialization Code

The initialization code is directly called by the bootcode, after the CPU is set up to execute C++ / C compiled code. The purpose of the initialization code is to initialize the hardware and software components, so the applicaton can be started when the initialization is done. Figure 3.3 shows the basic flow of different initialization steps, that are performed.

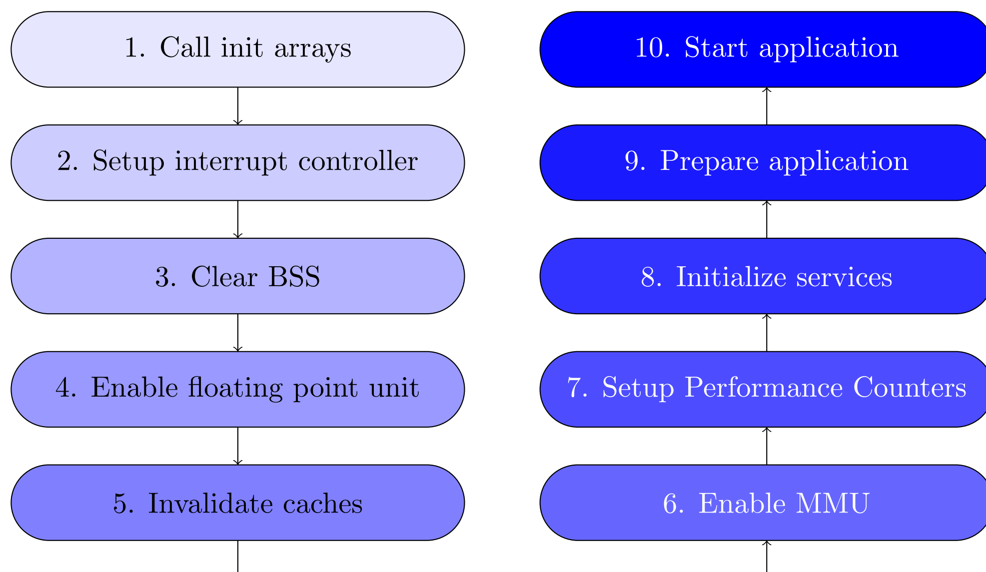

Figure 3.3: Steps of the Initialization Code

The detailed steps of the initialization code are the following:

1. **Call init arrays**: As the entire runtime system is compiled with the GNU gcc [3], the init array is created by the compiler. For C++ applications, global instances have to be created (i.e. the constructor has to be called on the object), before the application starts running. In a normal Linux application, this is handled by automatic generated code, but for a bare-metal program the very first executed code is already the provided bootloader. In fact, the constructors have to be called manually by the initialization code. The linker configuration of the bare-metal runtime system places the so-called init section at a central, well known location. The init section is a list of function pointers, generated by the gcc compiler, which perform the required object construction. The initialization code figures out the location, where the init array is paced and starts calling each function in the list. After this, global instances are created (for instance the debug log infrastructure) and can be used.

2. **Setup interrupt controller**: For modern CPU architectures, the handling of external interrupt requests is handled by a set of dedicated controllers. X86_64 CPUs usually integrate the Advanced Programmable Interrupt Controller (APIC) [19], while ARM based CPUs usually integrate the Generic Interrupt Controller (GIC) [7]. The workflow of both controllers is very similar. Both consist of a global interface, which is connected to all the physical interrupt sources (the IOAPIC for X86_64 and the GIC Distributor for ARM). This global interface is connected to CPU local interfaces for every CPU core (the LAPIC for X86_64 and the GIC CPU Interface for ARM). The global interface can be configured how to route interrupt requests to different cores, furthermore interrupt sources can be masked and unmasked. The runtime system implements a driver for the GIC Distributor and the GIC CPU Interface. During initialization both have to be set up and activated. During this process, each interrupt source is configured and masked / unmasked. The runtime system masks all interrupts by default. When an application or a system service requires interrupts, it has to call the driver to unmask them.

3. **Clear the BSS**: The Block Storage Segment (BSS) is used to store global, uninitialized data. To prevent errors, the memory BSS region is written to 0 during initialization, because it may contain any memory content. The linker configuration of the runtime system again places the BSS at a central, well known location, so the initialization code can write to it easily.

4. **Enable floating point unit**: By default, the floating point unit (and vector processing unit) is disabled on Cortex-A53 CPUs [8]. To enable the application to perform floating point and vector calculations, the corresponding unit has to be en-

abled. For Cortex-A53 no real enabling is required, by default every access to the floating point unit causes an abort trap, which has to be disabled.

5. **Invalidate caches**: Before the initialization code executes, the CPU internal caches are disabled and may contain invalid content from previous running bootloaders. This would cause an abort trap while enabling the caches, thus the content of the caches is invalidated first. The caches might be enabled after this, but currently the runtime system leaves caches disabled to analyze the memory behavior of the executed application.

6. **Enable MMU**: The Memory Management Unit (MMU) controls the translation of virtual to physical memory addresses and furthermore defines access permissions for memory pages on a granularity up to 4kB. To enable the MMU, a pagetable has to be set up, which contains one entry for every virtual page [1][4]. This entries hold the mapped physical page and the setting of access permissions for each virtual memory page. The runtime system sets up a pagetable with identity mapping (all virtual addresses correspond to their identity physical addresses) and enables the MMU afterwards. Additionally, the runtime system provides a system service to modify the pagetables later on. The pagetables can be modified regarding the physical mapping of pages and their access permissions.

7. **Setup performance counters**: The runtime system integrates a driver for the ARM Performance Monitoring Unit (PMU) [1], which allows architectural events (e.g. cache misses, cache hits, memory requests, ...) to be monitored and counted in special registers. During initialization, the driver is initialized and the number of available performance counters in the system is requested. Applications and system services can use the PMU driver later to keep track of the architectural events.

8. **Initialize services**: To enable software only wear-leveling, the bare-metal runtime system offers a set of system services, which are required. For the coarse-grained leveling, a write count approximation for virtual memory pages is recorded by the WriteMonitor service. For the fine-grained stack leveling, relocation functions for the stack and the setup of a shadow stack are handled by the StackBalancer service. These services are set up during initialization accordingly.

9. **Prepare application**: The target application of the runtime system runs isolated from the rest of the runtime system, thus a setup of the application is required. The `text` and `data` regions are configured with appropriate access and execution permissions and the stack is allocated and set up for the application.

[4]The pagetables don't need to have entries for every page strictly. Indeed, the pagetables are organized hierarchically, thus a block entry on a coarse granularity does not require a finer-grained pagetable for the corresponding memory region. In consequence, only the most coarse-grained pagetable has to be complete. The coarsest granularity is 512GB.

10. **Start application**: The application is already set up from the previous step. The runtime system does not simply call the first application function, because the application has to run on a lower privileged exception level (EL). This is necessary, since interrupts during the application execution should not influence the application. To achieve this, all interrupts are handled on the exception level of the runtime system, while the application runs on another exception level.

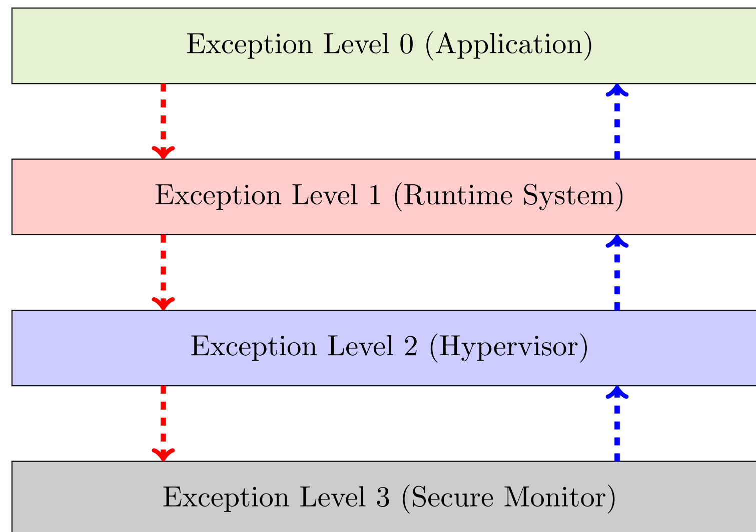

Figure 3.4: Exception Level Overview

Figure 3.4 illustrates the usage of different exception levels. Each exception level can call the upper exception levels with special call instructions. The application uses Supervisor Calls (SVC) to run services from the runtime system. The runtime system hands back to the application by finishing the SVC call procedure. The runtime system configures the CPU in a way, that each interrupt on the application level (EL0) are taken to EL1, thus the application is not influenced by the interrupts. The Hypervisor level (EL2) is not used by the simulation environment. The Secure Monitor (EL3) runs the ARM Trusted Firmware, which cannot be changed at all. The trusted firmware can be called with a System Management Call (SMC) to perform hardware actions (e.g. trigger a reboot, set power modes, ...). The runtime system boots into EL1 and calls the application on EL0 by faking the finishing of a SVC call procedure.

Debug Infrastructure

As the simulation mode of gem5 is the full system simulation mode, no simple output to the Linux console is provided. Of course, gem5 offers the possibility to connect an external debugger (i.e. gdb), but a simple text output is also required and sometimes more

useful. To enable this, the runtime system implements a device driver for the Universal Asynchronous Receiver and Transmitter (UART) controller [6], which is a common UART controller in many ARMv8 based SoCs. As the driver only allows to output a single character to the UART console, additionally a text formatting output stream implementation is provided, which is similar to the C++ stdlib output stream implementation. The output stream allows applications and the rest of the runtime system easily to output text messages, numbers, pointers, etc. to the UART console. Gem5 redirects the simulated output of the UART controller to a file, which can be monitored during the simulation.

OutputStream
+ instance : OutputStream + base : uint64_t
+ operator<<(string : const char *) : OutputStream& + operator<<(number : uint8_t) : OutputStream& ... + operator<<(number : uint64_t) : OutputStream& + operator<<(number : int8_t) : OutputStream& ... + operator<<(number : int64_t) : OutputStream& + operator<<(number : double) : OutputStream& + operator<<(ptr : void *) : OutputStream& + operator<<(manipulator : OutputStream& (*)(other : OutputStream&))) : OutputStream& + dec(other : OutputStream&) : OutputStream& + bin(other : OutputStream&) : OutputStream& + hex(other : OutputStream&) : OutputStream& + endl(other : OutputStream&) : OutputStream&

Figure 3.5: Class Overview of the Output Stream

Figure 3.5 gives an overview of the interface functions of the output stream, which can be called from any application. All operators return a reference to the output stream, thus multiple calls can be chained. The manipulator functions dec, bin and hex (which are not class members) can be passed as an argument to the manipulator function and will change the internal base attribute, which controls the formatting of numbers. An example to use the debug infrastructure is shown in Listing 3.1.

```
#include <system/service/OutputStream.h>

uint64_t variable=42;
uint64_t *ptr=&variable;

OutputStream::instance << "Variable␣" << dec << variable << "␣(binary:␣" << bin
<< variable << ")␣resides␣at␣" << ptr << endl;
```

Listing 3.1: Example of Using the Output Stream

Systemcall Interface

As already explained in Section 3.1.2, the application is executed on a lower exception level than the runtime system. In order to this, the application cannot simply call functions of the runtime system to request a system service. Instead, the application has to perform a Supervisor Call (SVC), which causes an interrupt, handled by the exception level of the runtime system. The runtime system has to provide an interrupt handler, which detects the Supervisor Call as the interrupt cause and triggers the according system service. The selection of the system service has to be done by a single number, which can be passed to the SVC instruction. The interrupt handler can look up the number and select the system service. To ease this process, the runtime system provides two components. One component, called systemcall interface, is part of the application and contains well named functions, according to the requested system service. The function itself executed the SVC instruction with the corresponding number. For instance, the systemcall interface to exit and shutdown the entire simulation is shown in Listing 3.2.

```
#include "syscall_interface.h"

void syscall_exit() { asm volatile("svc␣#0"); }
```

Listing 3.2: Systemcall Interface Function

The generated interrupt from the systemcalls is processed by the interrupt handling mechanism of the runtime system. In case the interrupt mechanisms detect a systemcall causing the interrupt, the passed number is extracted and a corresponding function of the second component of the infrastructure is called. This call happens on the exception level of the runtime system and can trigger different components of the runtime system to take action. Once the handling is done, the interrupt mechanism finishes the interrupt handling and returns to the application on it's own exception level. If more systemcalls are required, this infrastructure can easily be extended with new functions.

Drivers and Libraries

During the explanation of the initialization code, it was mentioned that several drivers are provided by the runtime system and have to be initialized during the initialization. The runtime system provides four device drivers and two library classes to enable the system services and the application to utilize the required hardware components. The device drivers are following:

1. **Generic Interrupt Controller** [7]: The driver for the GIC consists of two components, the driver for the global GIC Distributor and for the local CPU interface. Both components implement a procedure to initialize the controller and program it with default settings. This procedures are called during the initialization. Additionally, the distributor driver can mask / unmask interrupt sources, set interrupt

priorities, set the target CPU cores per interrupt and request the current configuration for every interrupt source. The driver of the CPU interface can filter interrupts from the distributor by a priority threshold. For handling interrupts, the driver supports to lookup the causing interrupt source and to signal the finishing of the interrupt handling.

2. **PL011 UART** [6]: The UART driver implements an extreme simple design. As the UART controller is usually already configured by the bootloader, the driver implements no configuration routines by now. The only purpose of the driver is to write a single character to the UART controller and make sure, it has been sent out.

3. **Performance Counter** [1]: The performance counters are a generic mechanism to count CPU internal events during execution. The driver supports to set up the counters with several settings and select the event, which is counted. Usually, the CPU has some special performance counters (e.g. the CPU cycle counter) and a set of general purpose counters, which can be programmed with arbitrary events. The driver is capable of setting up the cycle counter, as well as detecting the number of general purpose counters, read and write values to the counters, program the event to count, program filters for the event counting and enable the overflow interrupt generation for every general purpose counter. Whenever an interrupt from a performance counter occurs, the runtime system figures out the counter which caused the interrupt and triggers an according system service eventually.

4. **Memory Management Unit** [1]: The driver for the MMU is not only capable of setting up pagetables and activating the MMU (performed during initialization), but is also capable of modifying pagetable entries per virtual memory page. For every page, the physical mapping and the access permissions can be programmed. Every change in the pagetables requires a maintenance operation for the Translation Lookaside Buffer (TLB). As the pagetable entries are eventually cached by the TLB, a modification might take no action until the entry is reloaded into the TLB. To overcome this, each modification causes the TLB to invalidate the entries, that correspond to the modified entry. The maintenance operations are also provided by the MMU driver. Finally, the MMU driver implements routines to invalidate and activate the CPU caches.

The previously described device drivers are strongly required by the wear-leveling techniques in this thesis, but also some library support is needed. As the entire runtime system is compiled as a bare-metal application, the C and C++ standard libraries are not available to the runtime system, neither to the application. At least, some applications require floating point operations and one wear-leveling technique requires an Red-Black Tree (RB-

Tree) as a data structure. Because of these needs, the runtime system implements both features as a small library.

3.1.3 Performance Counter Extension for gem5

Section 3.1.2 describes in detail the runtime system, provided by this thesis. It is mentioned, that the performance counting mechanism of the CPU is required for the wear-leveling later on and thus, the runtime system provides drivers to use the performance counters. Unfortunately, gem5 does not simulate the Performance Monitoring Unit completely by default. Only a small subset of the events, which can be counted with the general purpose counters, is implemented. The wear-leveling mechanism in this thesis requires the `BUS_ACCESS_ST` event, which counts the number of store requests to the memory bus [2]. This event is not in the supported subset of events. To overcome this issue, the event is implemented in gem5 as a part of this thesis. Additionally, the PMU is not available in the default gem5 full system simulation configuration. This subsections points out how the PMU is enabled in gem5 and gives details about the implementation of the `BUS_ACCESS_ST` event.

Enable PMU

Gem5 mostly uses Python scripts for configuration purposes. Consequently, the assembling of the simulated system in done in central python configuration file. By default, already a `system` with a `cpu` as a member is created. The `cpu` class has a member `isa`, which has a member `pmu`. The `pmu` member can be assigned with an arbitrary PMU implementation, but is not assigned by default. The gem5 repository already provides a class ArmPMU, which contains the (limited) implementation of the required PMU. The system configuration script is modified to create an instance of the ArmPMU implementation and assign it to the `pmu` member. The ArmPMU implementation requires on it's own to be initialized, which is also done in the system configuration script. The configured members define the way of interrupt generation and the interrupt source number at the GIC.

Finally, the ArmPMU implementation requires the function `addArchEvents` to be called with some references to already created components of the system (members of the `system` class). This function creates an instance of a dedicated implementation for each event and adds it to the ArmPMU implementation.

Implementation of the `BUS_ACCESS_ST` Event

As already stated, the gem5 implementation of the PMU is highly modular and configurable. This allows another event easily to be added. Basically, two major modifications need to be performed:

1. An implementation of the counter event has to be provided. The implementation has to provide an instance of the class ProbePoints::PMU. This instance can notify all performance counters, which are set to listen to this event, to increase their value. The instance is registered with an identifying name string. The rest of the implementation can be arbitrary, only the ProbePoints::PMU instance has to be called, whenever the counter values should be increased.

2. The `addArchEvents` function need to be modified to assign the counter event to an event number. The event number is used by the software to select the event by writing the number into a special register. To register the new event, an additional call to the `addEvent` function has to be added. The function takes the event number, an instance of the event implementation and the identifying name string of the ProbePoints::PMU instance as arguments.

The implementation of the `BUS_ACCESS_ST` event is achieved as a simple implementation here. A class Nvmain_Write_PMU is added to the NVMain2.0 sourcetree (which is compiled together with gem5) and offers the function `triggerWrite`. The NVMain2.0 simulator is modified to call this function, whenever a write request reaches the simulator. The `triggerWrite` directly updates the internal ProbePoints::PMU instance, so the ArmPMU implementation can update the performance counters eventually. As an instance of the Nvmain_Write_PMU needs to be accessed from several other classes, a global instance is created by the system configuration script.

3.2 Application Separation

The allover goal of the previously presented simulation environment is to execute and analyze test applications. The applications should be analyzed especially regarding their memory access behavior. Later on, the memory access behavior should be compared to a version, where different wear-leveling techniques are applied. To analyze the pure memory behavior of the test applications, the executed application has to be separated from the rest of the runtime system in-memory. The runtime system itself accesses the memory from time to time (e.g. during interrupt handling), thus these memory accesses should be isolated from the applications accesses. Of course, in a realistic use case, the memory accesses from the runtime system cannot be ignored and have to be wear-leveled, too, but this thesis aims to point out the basic connection between program execution and resulting memory accesses. This can only be done on relative simple applications and without the interference of complex runtime system executions. For the realistic use case, the learned insights might also be applied on the runtime system later on.
However, the runtime system implements two techniques to separate and isolate the runtime system and the application in-memory. One is to place all accessed memory regions

at different locations (one for the runtime system and another one for the application) and the second technique is to force the stack, used for interrupt handling, to the stack of the runtime system. Both techniques are stated in detail in the rest of this section.

3.2.1 Spatial Separation

As already explained, the memory regions of the runtime system and of the application should be completely different. The build process of the runtime system compiles the runtime system and the application into one binary file, which is then loaded into the gem5 simulator. Usually, the gcc compiler would place all memory regions of the runtime system and the application together, organized in the `text`, `data`, `bss` and `stack` segment. The custom linker configuration of the runtime system separates the `text`, `data`, `bss` and `stack` segment from the application and places them to a memory region, which resides behind the `text`, `data`, `bss` and `stack` segments from the runtime system. Additionally, the linker configuration places symbols at the beginning and ending of all memory regions of interest.

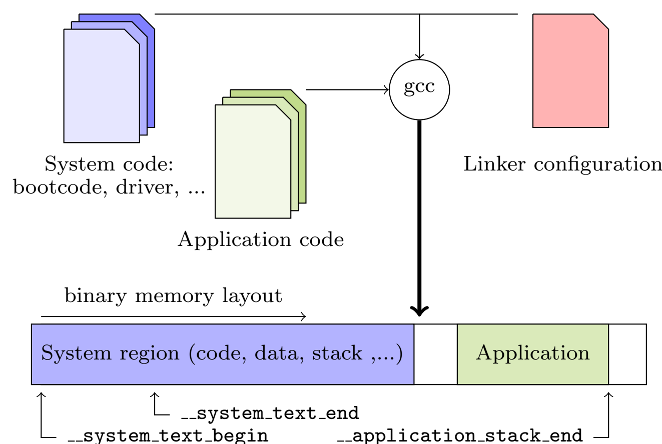

Figure 3.6: Memory Placement of the Application by the Linker Configuration

Figure 3.6 illustrates the compiling process and shows the resulting memory layout. The `stack` is not automatically allocated by the gcc, because the `stack` allocation ans setup is done during runtime. The linker configuration allocates two memory regions for the `stack`, one for the runtime system and one for the application. During boot and initialization, these regions are programmed into the stack-pointer register.

The symbols, which are placed at the beginning and ending of each memory region, can be either extracted from the compiled binary image or accessed in the runtime system. The extraction from the binary image is necessary for analysis purposes. The simulation setup produces a log file with accessed memory addresses, together with the information where each memory region begins and ends, the write count and write distribution for each

memory region can be determined. The memory symbols are also required during the runtime system execution, when for instance the access permissions have to be configured for a special memory region only (e.g. executable permission for the `text` region).

3.2.2 Interrupt Isolation

The technique, presented in Section 3.2.1 ensures that the memory accesses from the application will always target different memory locations than the memory accesses from the runtime system. This enables the memory behavior of the application to be analyzedt isolated from the runtime system. Unfortunately, the runtime system still might influence the write accesses to the application's stack. Whenever an interrupt is handled, the runtime system has to save the current CPU registers on the stack to restore them when the interrupt handling finished. Usually, the interrupt handler just uses the same stack pointer as the interrupted program, which might be the application. In order to this, the interrupt handling by the runtime system would write and read from the application's stack. This issue is resolved by using a different stack for the interrupt handling. As all the interrupts happening during application execution on EL0 are handled in EL1 anyway, the CPU can be configured to switch the stack pointer while changing the exception level. The stack pointer of EL1 is always configured to the stack of the runtime system, while the stack pointer of EL0 is always configured to the stack of the application. This ensures, every interrupt handling is completely processed on the stack of the runtime system and not on the stack of the application.

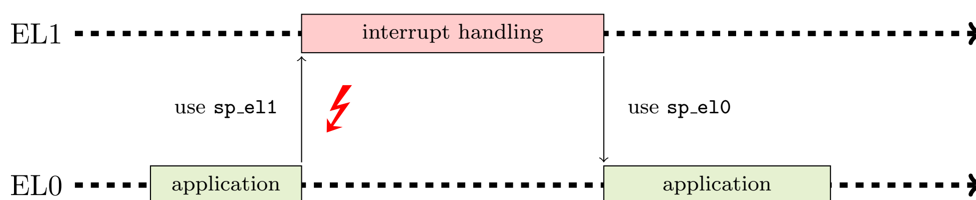

Figure 3.7: Exception Level Changing while Interrupt Handling

Figure 3.7 shows how the interrupt handling is processed. An interrupt, happening during the execution of the application on EL0 causes the hardware to use the stack pointer of EL1, called `sp_el1`. This stack pointer points to the stack of the runtime system, thus no matter what the interrupt handler does, the application stack is not affected. When the interrupt handling finishes, the exception level is changed back to EL0. Accordingly, the stack pointer is set back by the hardware to `sp_el0`, which points to the application stack. As the application stack is not touched at all during the interrupt handling, the application can continue the execution without restoring old values.

3.3 Program Region Analysis

With the help of the techniques described in Section 3.2, the memory accesses of an arbitrary test application target a dedicated memory region and are not influenced by the rest of the runtime system. Combining this with the memory traces of the runtime system, the memory access behavior of the application can be analyzed after performing a simulation run. To do so, the trace files are processed and the write count for each memory byte / cach-line is aggregated. With the help of the linker symbols, the memory areas of the application can be separated and furthermore, the memory areas of the application can be split into the different memory regions (`stack`, `text`, `data` and `bss`). Once the processing of the data is done, the memory accesses can be analyzed in different ways:

- Graphical illustration
- Analytical calculations
- Input to several physical models

While the graphical illustrations help to analyze the connection between the memory regions and a memory write pattern (e.g. like Figure 1.6), the analytical calculations can be applied to evaluate the write behavior regarding memory lifetime considerations (Section 1.2.1). In this thesis, the aggregated information is not passed to any further physical models, but theoretically this is possible. Additionally, not the aggregated data but the entire memory trace can be passed to detailed physical models to simulate a realistic memory behavior.
However, in this thesis the memory trace is usually illustrated graphically and the achieved endurance (AE) is calculated (Equation (1.4)) to evaluate a concrete simulation run. For now, this evaluation is processed on four typical benchmark applications:

- **bitcount** is a simple application, which counts the 1 bits in a big memory region. The application is implemented iterative, thus the memory region is processed in a long running loop.
- **fft** is a recursive implementation of the fast fourier transformation. The recursive implementation splits the data array in each run into two, half sized, arrays, one with all elements with even indexes and another one with all elements with odd indexes. This requires temporary arrays to be set up before each subsequent function call. These temporary arrays are placed on the caller's stack.
- **lesolve** is a solver for linear equation systems, applying the gaussian elimination algorithm. This application is implemented iterative and processes the array of equations in a long running loop.

- **qsort** is a recursive quicksort implementation. The array of numbers to sort is a global array and the elements are swapped within this array.

Running these applications in the simulation environment and separating the application's memory region, leads to memory traces, illustrated in Figure 3.8.

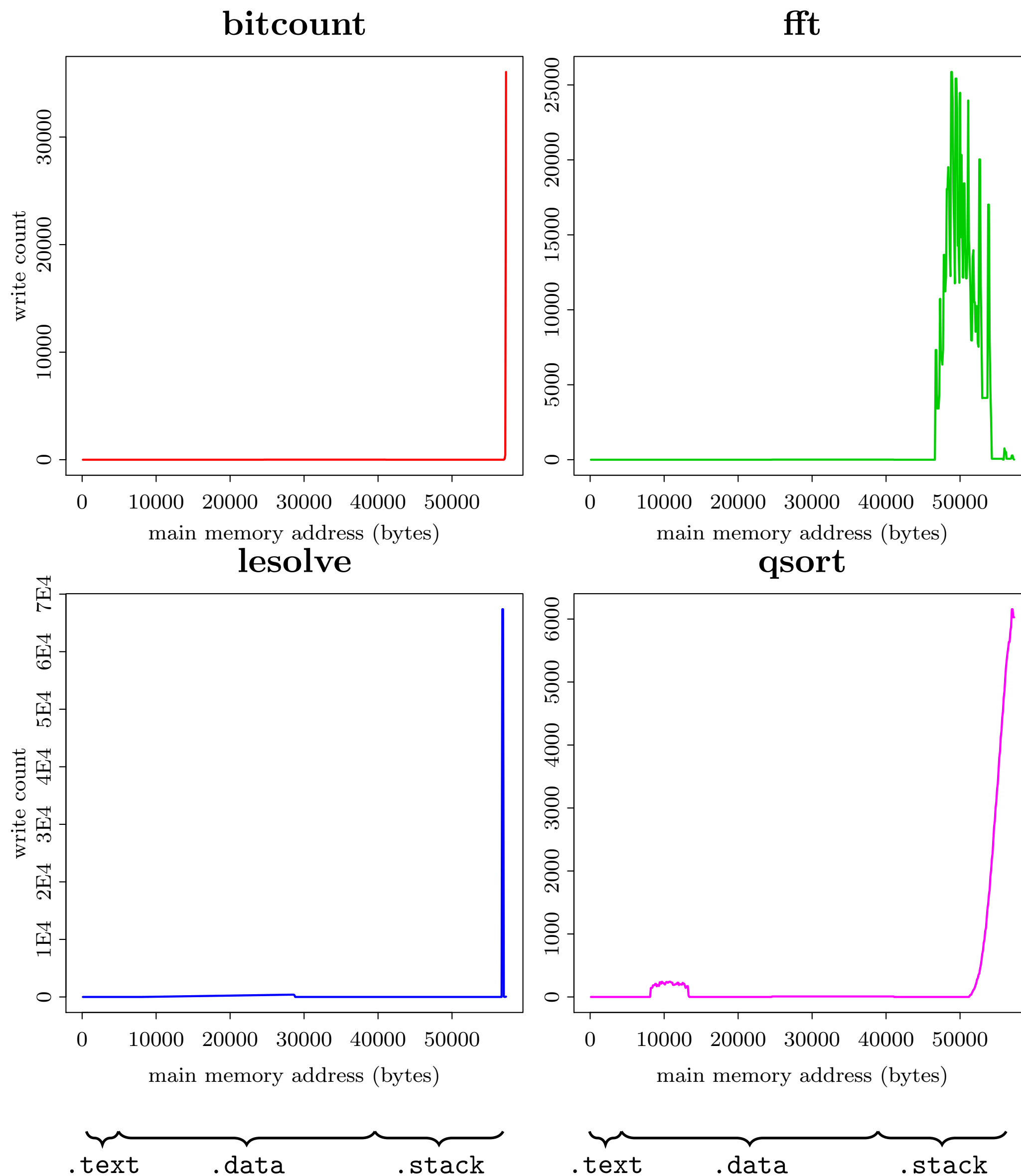

Figure 3.8: Memory Write Trace of Four Benchmark Applications

The illustrated memory traces can be connected to the program behavior:

- **bitcount**: As the application runs only a single loop over a big data array, which reads the numbers out of the array, no write accesses are performed to the `data` segment. The only written variables are the intermediate counting result and eventually intermediate results inside the loop. These variables are local variables and

reside on the stack. The location of these variables never changes, thus only a small region on the stack is written intensively.

- **fft**: The recursive implementation makes use of a lot of stack memory for two reasons. First, recursive calls lead to stack frames being created for each function call, which reside on the stack. Second, the temporary arrays are created for each function on the stack, thus each function makes use of a big part of stack memory. All together, the application uses a lot of the stack memory very intensively. The data itself is never written, thus only the stack is written.

- **lesolve**: The memory write trace looks similar to the write trace of **bitcount**, because both are applications with only one long running loop. The majority of write accesses targets local intermediate variables on the stack, which reside in a very small region of the stack. As **lesolve** solves the equation system in place, the `data` segment is written.

- **qsort**: In comparison the recursive **fft** implementation, the **qsort** application does not use big memory regions locally on the stack. Each function call only requires some local variables for intermediate results. The usage of the stack anyway targets a big memory area, because of the recursive function calls. The stack frames for the functions take some additional space, the recursion depth determines the size of the used stack memory accordingly. Again, the array is sorted in place, which leads to write accesses to the `data` segment.

Summing up, the graphical illustrations show that the memory of the application is written very uneven by the application. It can be observed, that there is a big difference in the usage of the stack memory for recursive and iterative implementations. The write pattern strongly depends on the application itself in two ways. The application logic might cause writes to some memory regions, because variables are placed there. Additionally, the compiler generates write accesses, for instance for stack frames for functions, which depend on the application, but is not actively controlled by the code.
However, with respect to the memory lifetime of low endurance memory, the write behavior observed is a bad behavior. Table 3.1 lists the calculated achieved endurance for the four benchmark applications. A number of 100% would mean, the application uses the memory perfectly, regarding the memory endurance. The numbers show, that all the applications achieve a really bad endurance, due to their uneven write pattern.

	bitcount	fft	lesolve	qsort
Achieved endurance	0.12%	5.1%	0.23%	3.04%

Table 3.1: Achieved Endurance for Four Benchmark Applications

4 Page Based Memory Relocation

To tackle the problem, which can be deferred from Table 3.1, several solutions have been proposed in literature (Chapter 2). While some solutions propose software only solutions, e.g. aging-aware memory allocators [22, 28, 21], other solutions propose a combination of hardware and software to keep track of the aging in hardware and trigger rebalancing actions in software or in hardware [14, 30]. The rebalancing actions usually aim to modify a logical to physical memory mapping[1] to redirect the writes from the running software, without any corporation of the software. The decision, which memory region is remapped to which location, usually is made by a wear-leveling algorithm.

As the software only solutions usually depend on an application support (e.g. sufficient memory allocations need to be performed) and do not introduce an automated, relocation based technique, they suffer in certain scenarios. For instance, in an embedded system an application might allocate memory at the startup and use it until shutdown to write content into it. Without an automated remapping technique, the memory where the allocation was performed ages faster than other memory regions. The proposed hardware based or hardware / software based solutions overcome this issue, but require special hardware support (at least a counting mechanism for memory writes to make aging-aware decisions). Up to now, it is unclear if this hardware support can be realized fast and efficient enough to be provided even in small embedded systems. Currently available memory chips with non-volatile memory, usually do not support write count tracking. Most of the currently available memory controllers do this neither.

The disadvantages of both approaches motivate another approach, which is provided by this thesis and is described in this chapter. The proposed approach performs aging-aware, automated relocations, without the need for special hardware support. The approach only uses commonly available hardware and performs the rest of the wear-leveling in software, thus it is called "Software Only Endurance Leveling". The limitation on commonly available hardware causes two major design decisions:

- The relocation of memory regions is performed by the virtual to physical memory mapping of the MMU. Thus, the granularity of remapped memory regions is bound to the paging granularity of the MMU, which is 4kB in the case if this thesis.

[1] This approach is not limited to use the mapping mechanisms of the MMU. Some proposed works introduce another mapping mechanism on another granularity in hardware.

- To make the relocation decisions aging-aware, the write count to the different virtual memory pages is approximated by a write count sampling. This can be done without additional hardware support, on the cost of a CPU overhead and accuracy. However, the evaluations (Section 4.3) show that the approximation is precise enough to enable good aging-aware relocation decisions, while causing a considerable CPU overhead.

This chapter describes the "Software Only Endurance Leveling" approach in detail. First, in Section 4.1 the approximation mechanism for the write counts, called endurance tracking, is described. This mechanism is not bound to a concrete wear-leveling algorithm, it aims to enable arbitrary aging-aware wear-leveling algorithms. The endurance tracking mechanism can be extended to respect physical properties of the underlying memory, so the wear-leveling algorithms does not have to respect them. Section 4.2 introduces an example wear-leveling algorithm, which is used for the evaluations. This wear-leveling algorithm sorts the virtual memory pages according to their (approximated) write counts and relocates heavy written pages to the currently least often written page. A detailed evaluation with the 4 benchmark applications is provided in Section 4.3. First, the accuracy of the endurance tracking mechanism is evaluated without combined wear-leveling. After this, the wear-leveling algorithm is activated with the approximated write counts and the achieved endurance with wear-leveling is evaluated. After discussing the several overheads, the provided wear-leveling approach is compared to a hardware based aging-aware wear-leveling algorithm from the literature. Finally, the discussion in Section 4.4 points out the advantages and drawbacks of "Software Only Endurance Leveling". Especially, the usage scenario and possibilities to hide the overhead are discussed.

4.1 Software Only Endurance Tracking

As already explained, this thesis proposes an endurance tracking mechanism, which does not require special hardware and enables aging-aware wear-leveling algorithms, which usually require a write count from special hardware extensions. As there usually is no hardware, which allows to count the write requests to different memory regions, the goal of endurance tracking cannot be achieved by a special configuration of the existing hardware. Indeed, the endurance tracking system does not provide a precise write count to different memory regions, but only a statistical approximation of the write distribution. As long as this distribution is accurate enough, aging-aware wear-leveling algorithms can use the approximation as an input and balance out the write count over different memory regions. It is not the scope of this thesis to provide a good wear-leveling algorithm or to compare different wear-leveling algorithms, but to enable aging-aware techniques generally with the write approximation. Technically, already existing aging-aware wear-leveling algorithms can be supplied with the write count approximation from the endurance tracking system, instead of a hardware provided write count.

In this Section, first the technical details about the endurance tracking system are presented. After this, the ability to respect additional physical models (if the write count not directly corresponds to endurance) is discussed. In the end, the overheads, caused by the endurance tracking system are discussed.

4.1.1 Write Distribution Recording

To record a statistical write distribution and use it as an approximation for the write count, two main techniques are used in this thesis. The first technique is a write count sampling. This means, the total number of writes to the entire memory is counted by a CPU integrated performance counter (Section 3.1.3) and when the value reaches a certain threshold, an interrupt is triggered. Usually, common CPUs allow to trigger an interrupt once a performance counter overflows[2]. To cause an overflow interrupt after exceeding a certain threshold, the performance counter register has to be programmed with the maximum value minus the certain threshold. If this reprogramming is performed on every overflow interrupt, the interrupt occurs whenever the counter is increased by the certain threshold by the hardware.

```
void WriteMonitor::handle_pmc_0_interrupt(uint64_t *saved_stack_base) {
    ...
    PMC::instance.write_event_counter(0, UINT32_MAX - MONITORING_RESOLUTION);
}
```

Listing 4.1: Performance Counter Reset

Listing 4.1 is an excerpt from the implementation of the endurance tracking system. The function `handle_pmc_0_interrupt` is called from the interrupt handling mechanism of the bare-metal runtime system, whenever the performance counter with the number 0 causes an overflow interrupt. The handler sets the value of the performance counter with the number 0 to the maximum value minus the threshold.

Using this configuration of a performance counter, the real write distribution from the running application is interrupted after an equal number of performed writes. If the system records the last written memory address on every interrupt, the result is a sampled distribution with equal weights for all memory regions. Unfortunately, the last written memory address cannot be recorded simply, because depending on the hardware implementation of the performance counter, the interrupt might not interrupt the write instruction, which caused the overflow of the performance counter. Thus, the written address cannot be determined by an analysis of the interrupted program counter. To overcome this problem, the endurance tracking system makes use of a second technique, which is called write access trapping.

[2] On a 64 bit CPU, the performance counter registers usually have a width of 64 bit or at least 32 bit. To overflow, the counter has to exceed the number of 18.446.744.073.709.551.615 or 4.294.967.295 (32 bit)

Write access trapping means, that the interrupt handler for the performance counter overflow does not record the last written address, but sets the MMU up in such a way, that the next write access to any memory region causes a permission violation trap, which has to be handled by the runtime system. This is achieved by setting the access permissions of the monitored memory regions to not writable. Once the application tries to write a not writable memory region, an access violation trap is caused and handled by the runtime system. For an access violation trap, the trap handler can directly extract the causing memory address from a dedicated register. The trap handler passes the information to the endurance tracking system, which increases the approximated write count for the corresponding memory region. The trap handler resets the access permissions afterwards[3], so the application can continue to execute. Applying this scheme, the written memory address of each nth write is recorded. This results in a precisely sampled write distribution of the application.

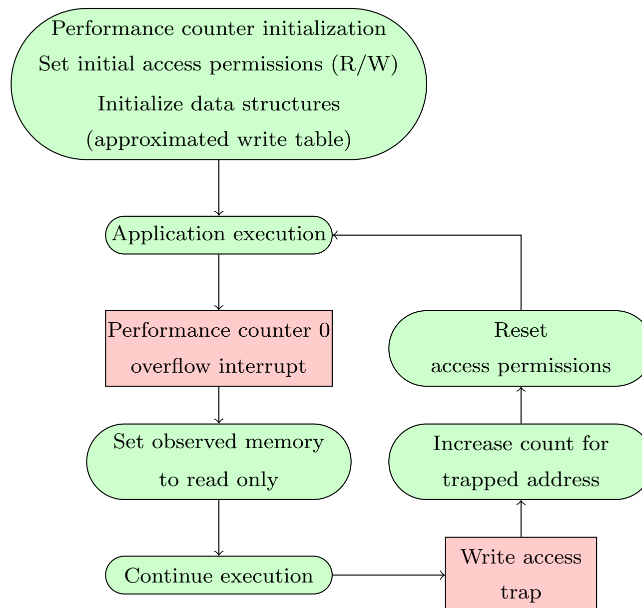

Figure 4.1: Workflow of the Endurance Tracking System

Figure 4.1 illustrates the flow of the write count estimation as described above. The maintained table with approximated write counts can be either read asynchronously by the wear-leveling algorithm, or the wear-leveling algorithm can be integrated into the endurance tracking system and can be called on an update of the approximated write count table.

[3]The bare-metal runtime system does not make any additional use of the access permissions, thus they can be simply set and reset. If an operating system requires the access permissions for other purposes, the corresponding permission at the end of the trap handler has to be determined again or has to be restored from a previous taken backup.

Limitations of the Endurance Tracking System

The implementation of the endurance tracking system permits to observe and track at least one memory region. As the violation of the read only access permission is handled by a trap handler, the trap handler uses at least some stack memory to process. During the trap handling, this stack memory has to be writable by the trap handler, because otherwise the trap handler would be always triggered again and never complete. As memory regions, which are tracked by the endurance tracking system need to be configured read only from time to time, the stack of the runtime system (where the trap handler executes) cannot be tracked. As the experimental work of this thesis focuses only on the memory of a test application, this does not give any drawback, but in a realistic setup the memory regions from the runtime system also have to be tracked and wear-leveling also has to also be applied. This issue could be overcome by a precise estimation of the performed write count to the stack memory. As the implementation of the trap handler is fixed and well known, the write count could be estimated by checking the taken branches. This estimation could be passed to the approximated write count table instead of the information collected by the tracking system for the memory regions of the stack of the runtime system.

The entire purpose of the endurance tracking system is to provide a statistical approximation of the write distribution of the running application to enable a wear-leveling algorithm to perform aging-aware decisions. Theoretically, the granularity of the tracking system, regarding time granularity and space granularity, can be configured arbitrarily. The performance counter overflow could be programmed to happen after one write access, thus every write access from the application would be trapped by the tracking system. In consequence, the real write distribution would be recorded and not sampled statistically. Additionally, the address, which caused the memory access violation, is available to the trap handler as a byte exact address. Thus, the approximated write table could hold an entry for every byte of the observed memory. Of course, configuring such a fine granularity causes a huge overhead. If every write access from the application is trapped, the trap handlers execute for every write access. This causes a high CPU overhead. If the approximated write count table holds an entry for every byte (e.g. at least a 4 byte counter value), the table consumes a lot of memory space. Fortunately, these overheads can be reduced by changing the granularities of recording. If the performance counter only overflows after a certain number of writes, the access violation will also occur less often, because it only occurs once after the performance counter overflow. This decreases the introduced CPU overhead. The approximated write count table could also only hold entries for equal sized memory blocks, which aggregate the accesses to all bytes within the blocks. This reduces the required memory space for the table. Thus, on the cost of accuracy of the tracked write distribution, the overhead can be configured arbitrarily.

4.1.2 Abstraction of Physical Properties

The previous section stated the technique of approximating the write distribution of a running application without the need for special hardware in detail. As already mentioned, some types of non-volatile memories do not age equally on every write, sometimes even memory cells of the same write age differently. The different aging can be caused by multiple reasons. For instance, PCM cells can suffer from a quality variation, called process variation [29]. Finally, this results in cells being programmed with different currents and thus, aging differently on every write. If a systems aims to achieve the maximum possible endurance on such a system, the process variation has to be respected and the strong cells have to be written more often than the weak cells. Up to a certain level, such physical models can be integrated into the endurance tracking system and can be abstracted to the wear-leveling algorithm. The basic steps are explained subsequently with the example of process variation.

During production, the quality of the cells is determined and the programming current is chosen according to the cell quality. Additionally, this information can be made accessible for the operting system by putting it into a read only memory (ROM) region. Doing this, the operating system knows, which cells or group of cells are programmed with which current and can determine the maximum write count for each cell according to a defined model. Knowing the maximum write count, the operating system can compute an aging factor for the different cells or group of cells. For instance, if one group of cells endures 1000 writes and another one endures 1500 writes, the first group would get a factor of 1.5, while the second group would get a factor of 1. This factor is multiplied with the approximated write count by the endurance tracking system and passed to the balancing algorithm afterwards. As the balancing algorithm aims to apply the same number of writes to all memory cells, it would apply mostly 1.5 times more writes to the second group, because the number is growing more slowly. Under the assumption, that the wear-leveling algorithm balances the write count incrementally and tries to reach an even balance at any point in time, the maximum endurance with respect to process variation could be achieved this way.

Obviously, the abstraction of physical properties only can work if the granularity of the physical property is equal or coarser than the spatial granularity of the endurance tracking system. For instance, if the endurance tracking system counts writes on the granularity of 4 kB virtual memory pages, it cannot respect a physical property which differs on the granularity of less than 4 kB. The example of process variation is not the only case, which could be handled by this technique. Any physical property, which can be reflected by a factor, which is multiplied with the approximated write count, can be respected using this technique. As the interface to the wear-leveling algorithm does not change at all, this technique is transparent to the wear-leveling algorithm. It balances out the writes to the

memory, respecting the physical mode, without actively knowing the physical model at all.

4.1.3 Granularity and Overhead Considerations

The previous sections point out, that there are several advantages and disadvantages when using different spatial and temporal granularities for the endurance tracking system. Choosing the best configuration might be a difficult task, but this subsection gives an overview, how the granularity might be chosen. First, the spatial granularity highly depends on the later applied wear-leveling algorithm. If the algorithm performs relocations only on a certain granularity, it is sufficient to estimate the write count on this granularity. A finer-grained write distribution cannot be respected, because relocations will never be performed on a finer granularity. For the wear-leveling algorithm in this thesis, the spatial granularity is configured to 4 kB, because the relocations are performed with the MMU on a 4 kB granularity. The memory overhead for the approximated write count table can be calculated easily. If 4 GB of memory are observed on a granularity of 4 kB and each entry consists of a 4 byte counter, the entire table requires 4 MB. Usually, this amount of memory should be a considerable overhead in a system with 4 GB main memory. However, if the spatial overhead is too high, the granularity still can be reduced and the overhead saved. This affects the quality of the write count estimation and thus the quality of the wear-leveling.

The second configurable granularity is the temporal granularity. For this granularity no value can be deduced from the wear-leveling algorithm, but from a knowledge about the application. If the application performs most of the writes to a small memory region, a coarser temporal granularity still captures most of the written memory regions. If the application writes a lot of memory regions in big time intervals in contrast, a coarse time granularity misses a lot written memory regions. If some knowledge about the write behavior of the application exists, this could be used to decide for a temporal granularity. If this knowledge is not available, it might be collected during runtime. If the currently recorded write distribution seems to be scattered over large memory regions, the runtime system could try to increase the temporal granularity dynamically and test, if the approximated distribution contains more details. The same technique works vice versa, if the currently recorded write distribution seems to target only a small memory region. The runtime system could try to decrease the temporal granularity in that case and test, if the approximated write distribution losses details. Such a dynamic technology is not implemented in the runtime system right now, but it could help to make a goof decision for the temporal granularity. The CPU overhead regarding the recording of the write distribution is directly determined by the temporal granularity (i.e. the frequency of the performance counter overflow). If the frequency is divided by 2, only half the number of

records have to be captured and the additional CPU overhead only introduces half the number of additional cycles. In an embedded system, real-time or performance constraints might limit the temporal granularity to an upper bound, due to the CPU overhead.
For both granularities, not only the introduced memory and CPU overhead might be interesting, but also the changed write behavior of the runtime system. For the evaluations of this thesis no wear-leveling of the runtime system is performed, but it would be in a realistic scenario. The execution of the endurance tracking system itself requires wear-leveling, because a lot of memory writes are spent in the trap handlers on the stack. The finer the granularities are configured, the more writes are performed by the tracking system and the more wear-leveling has to be done for the tracking system. This would introduce an additional overhead.

4.2 Online Endurance Balancing Algorithm

Up to here, the described endurance tracking system is completely independent from the applied wear-leveling algorithm. To evaluate finally, if the approximated write distribution is precise enough to enable aging-aware wear-leveling, the endurance tracking system has to be executed together with a lightweight wear-leveling algorithm. Doing this, the resulting memory trace point out, if the writes from the application are equally balanced by the entire setup. This section describes the implementation of the wear-leveling algorithm, used for the evaluation.
Regarding to the literature, different wear-leveling algorithms and data structures have been proposed to make aging-aware relocation decisions. Usually, these algorithms try to exchange the physical location of heavy written (hot) and less often written (cold) memory regions. The memory regions are typically managed in a data structure, which is optimized to identify the corresponding hot and cold pages quickly. Following this idea, the wear-leveling algorithm in this thesis also uses a data structure to manage memory regions (i.e. virtual memory pages) with respect to their (approximated) write count. The data structure supports a fast identification of the the least often written page and supports efficient updates. Section 4.2.1 describes the implementation of the data structure and how the virtual memory pages are managed within this data structure. After this, the integration of the wear-leveling algorithm and the endurance tracking system is described. Finally, Section 4.2.3 details the implementation of the relocation routine.

4.2.1 RBTree Based Page Management

As the management data structure, the wear-leveling algorithm implements a Red-Black Tree (RBTree) [10], which is ordered according to the approximated write count for each virtual memory page. A RBTree basically is a binary tree, which is guaranteed to be balanced. This is achieved by maintenance operations at insertion and deletion of nodes,

which use a binary coloring of each node. A set of rules for color sequences in the tree ensures the balance. As the tree is balanced, a node (i.e. the current minimum) can be determined with logarithmic runtime. Additionally, insertion and deletion are also fast. At the setup of the wear-leveling algorithm, all balanced virtual memory pages are inserted into the tree. Once a relocation is triggered for a virtual memory page, the current minimum (the least often written page) is extracted from the tree. Both pages are exchanged (the hot page is remapped to the cold page and vice versa) and the approximated write count for the former cold page is increased. The tree is not sorted directly among the write counts provided by the tracking system, because every update would require a reordering of the tree. Instead, the tree maintains a secondary counter for each virtual memory page, which is only modified during a remapping of the page. As remappings are triggered whenever the approximated write count from the tracking system exceeds a certain threshold, the secondary counter reflects the write distribution precisely. After the former cold page is reinserted into the tree[4], the execution can continue.
Using this technique, the approximated write count to each virtual memory page is balanced incrementally. Whenever the page is determined as the least often written page, it is reinserted with an increased approximated write count. Thus, all pages become the currently least written page at a certain time and the approximated write counts are always close together.

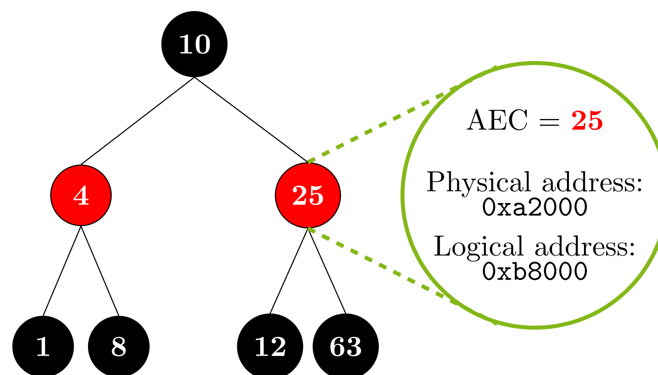

Figure 4.2: Organization of the Virtual Memory Pages in a RBTree

Figure 4.2 illustrates the management of the virtual memory pages within the tree. The Assumed Endurance Count (AEC) is the secondary counter, maintained by the tree. Each node stores three pieces of information about the virtual memory page. First the AEC, which is used to order the tree and additionally the logical and physical address. While the logical address never changes and only is used to a identify a page inside the tree, the physical address is updated, whenever a remapping is performed with the corresponding

[4] The former hot page does not need to be reinserted, because the approximated write count did not change. The only modification is the physical memory mapping. As this does not affect the structure of the tree, it can be updated without removing the corresponding node from the tree.

virtual memory page. The storing of the physical address in the RBTree saves the lookup of the physical address in the virtual memory pagetable. The tree is implemented in a way, that the nodes are placed in a consecutive array, ordered among their virtual memory addresses. The structure of the tree is provided through child pointers in every node. The placement of the nodes in this array, allows to change the physical address information of a known virtual memory page, without searching or touching it inside the tree. This is used to update the physical mapping of the page, that caused the relocation, because only the AEC of the target page is modified and thus, only the target page needs to be extracted and reinserted into the tree.

4.2.2 Integration in the Tracking System

The previous subsection explains the detailed usage of a RBTree to manage the virtual memory pages according to their current write count. The wear-leveling is achieved by exchanging hot pages with the currently least often written page, which is extracted from the tree. It is also explained, that the wear-leveling algorithm maintains its own endurance counter for the virtual memory pages, thus the integration of the wear-leveling algorithm into the endurance tracking system is very simple. As the endurance tracking system anyway is driven by interrupts (performance counter overflows and access permission violations), the wear-leveling algorithm is called directly during the interrupt service routine (ISR) of the access permission violation interrupt. During the ISR, the endurance tracking system increases the approximated write count for the faulting virtual memory page. Once the approximated write count exceeds a configured threshold, the wear-leveling algorithm is called. The wear-leveling algorithm extract the target page from the tree and remaps the faulting and the target page. The assumed endurance count of the wear-leveling algorithm is adjusted and the interrupt handling finishes. The threshold, which has to be exceeded to call the wear-leveling algorithm, is another configuration option, which can control the overhead for page relocations. By reducing the overhead, respectively increasing the threshold, less relocations are performed and thus, the achieved endurance of the wear-leveling might be worse.

The integration of the wear-leveling algorithm of this thesis into the endurance tracking system requires a certain corporation of the endurance tracking system and the wear-leveling algorithm. The tracking system keeps track of the threshold and calls the wear-leveling algorithm only when the threshold is exceeded. Accordingly, the wear-leveling algorithm has to provide an interface to the endurance tracking system, which can be called when the threshold is exceeded and the faulting address can be passed. This implementation saves some overhead, but is somehow specific to the used algorithm. Technically, the endurance tracking system can offer a more generic interface, to enable arbitrary wear-leveling algorithms to use the approximated write count. The most simple interface would

be to maintain a table with the approximated write count at a central memory location, thus a wear-leveling algorithm can read out the information and take action. It could be also possible, to allow the wear-leveling algorithm to implement an observer interface, which is notified when the table is updated. However, as it is not the scope of this thesis to run different wear-leveling algorithms on top of the endurance tracking system, this interface is not implemented.

4.2.3 Relocation Implementation

The RBTree based wear-leveling algorithm is notified about a hot page and selects a target cold page to exchange the hot and the cold page. This exchanging requires two actions. First, the virtual memory mapping has to be adjusted in a way, that the virtual memory page of the hot page is mapped to the physical location of the cold page and vice versa. Once the logical remapping is done, the TLB maintenance has to be performed to ensure the new mapping is applied by the MMU.

```
MMU::instance.set_page_mapping(vm_page, (void *)target.phys_address);
MMU::instance.invalidate_tlb_entry(vm_page);
MMU::instance.set_page_mapping((void *)former_vm, (void *)physical_address);
MMU::instance.invalidate_tlb_entry((void *)former_vm);
```

Listing 4.2: Virtual Address Remapping

Listing 4.2 is extracted from the implementation and illustrates how the adjustment of the logical memory mapping is performed. The `vm_page` holds a pointer to the virtual address of the memory page, which is hot and has to be relocated to a cold page. `target` is the cold page, which is extracted from the tree. `physical_address` is the physical address, which was mapped to the hot virtual memory page originally. It can be observed in the code, that the TLB entries, corresponding to the virtual address, are invalidated after the mapping of the virtual page is changed.

Secondly, the content of both memory pages has to be exchanged to maintain the perspective of the running application. The application still expects the data to be placed at a certain virtual memory address. By changing the physical mapping and also the physical content, this is ensured. The copying of the data itself requires a spare memory location as a buffer. This buffer location could be only the size of a single byte, or the size of an entire virtual memory page. In this setup, the size of the buffer is set to the size of a virtual memory page for two reasons. First, the copying of a big, consecutive memory region can be done faster and more efficient in most hardware architectures than copying single bytes from alternating locations (hot page to buffer, cold page to hot page, buffer to cold page). Secondly, wear-leveling should be also applied to the buffer region in a mature setup, because it is written on every relocation. If the buffer has the size of a virtual memory page, all memory cells within the buffer are written equally often, which achieves

a perfect wear-level within the page. The page based relocation can remap the physical location of the buffer from time to time, according to an approximated write count[5].

4.3 Evaluation

The previous sections of this chapter point out the basic concept as well as the implementation detail of the endurance tracking system and the wear-leveling algorithm. To state the quality of these approaches, evaluations are performed with four different benchmark applications. In this section, the same benchmark applications as in Section 3.3 are used to evaluate the approaches. Generally, all evaluations are executed in the simulation environment (Section 3.1), analytically evaluated according to Section 1.2.1 and graphically analyzed with the tools, described in Section 3.3. For every evaluation run, the configured parameters of the endurance tracking system and the wear-leveling system are described and the results are discussed. In this section, first the endurance tracking system is analyzed regarding the approximation of the real write count distribution in Section 4.3.1. After this, the wear-leveling algorithm is executed on top of the endurance tracking system to evaluate, if the approximated write count is detailed enough to enable aging-aware wear-leveling (Section 4.3.2). Section 4.3.3 gives an overview about the overheads, caused by the tracking system and the wear-leveling algorithm. It is also discussed, how the overhead can be hidden. Finally, this section concludes with a direct comparison of the wear-leveling algorithm combined with the endurance tracking system to a fine-grained, hardware based wear-leveling approach in Section 4.3.4.

4.3.1 Endurance Tracking

As already has been pointed out in detail, the endurance tracking system can be configured regarding the spatial and temporal granularity. Because of the MMU based wear-leveling algorithm, the spatial granularity doesn't need to be finer than the size of virtual memory pages, because the algorithm anyway only relocates entire virtual memory pages. Setting the granularity to multiples of virtual memory pages is possible, but not considered in this evaluation. The reason is that the real write distribution from the benchmark applications (Figure 3.8) already shows a high non-uniformity within virtual memory pages. Thus, relocating virtual memory pages only can not achieve perfect results anyway. This effect would be increased by choosing a courser granularity on purpose. Finally, the spatial granularity for the endurance tracking system is always 4 kB in all evaluations.

[5] The remapping of the buffer requires no buffer space, because the buffer region contains no content, which has to be kept. The buffer region does not even has to be monitored by the endurance tracking system, because the number of writes per relocations is well known. The number of relocations can simply be counted and thus, the approximated write count for the buffer region can be determined precisely without the endurance tracking system.

The temporal granularity, respectively the threshold after which amount of write accesses a new record should be captured, might be configured to an arbitrary value. Setting this value to extreme low values causes a high CPU overhead while the accuracy might not be increased that much. In this evaluation, the temporal granularity is set to a threshold of 1000 write accesses and of 5000 writes accesses. For both configurations, the benchmark applications are executed like in Figure 3.8, but the endurance tracking system is activated. The endurance tracking system does not trigger the wear-leveling algorithm at all, thus only the approximated write distribution is recorded. Once the benchmark finishes the execution, the collected write distribution approximation is printed out. These results are graphically illustrated in Figure 4.3.

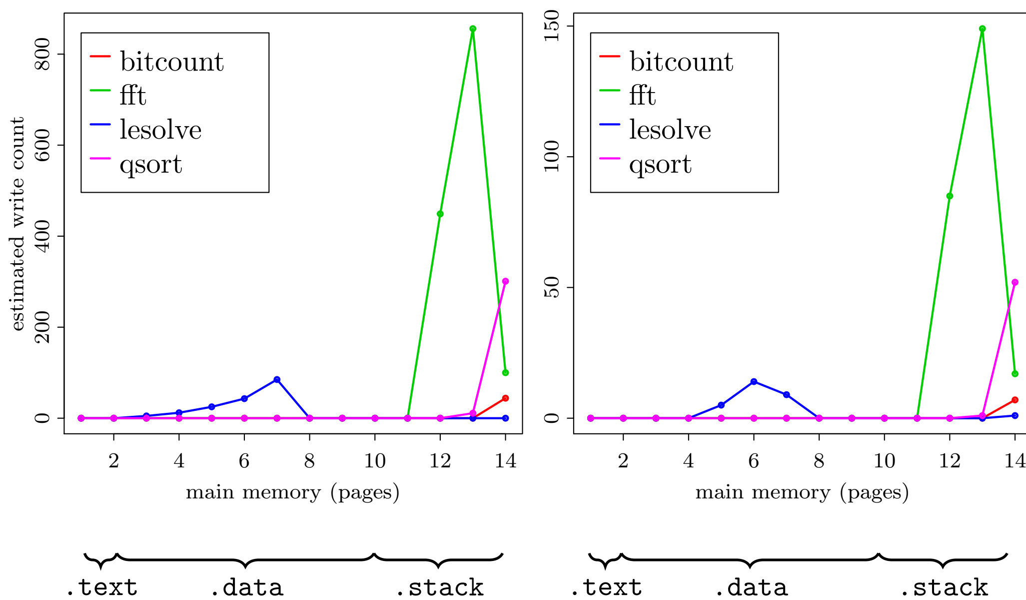

Figure 4.3: Approximated Write Distribution - 1000 Writes Threshold (Left) and 5000 Writes Threshold (Right)

It can be observed in the figure, that the basic characteristics of the write distribution is reflected in the approximation for a threshold of 1000 writes cycles as well as for a threshold of 5000 write cycles. Compared to the real write distribution of the benchmark applications (Figure 3.8), extreme small and high peaks are reflected less properly, because the absolute number of writes causing these peaks is low. This can lead to a loss of the peak during the sampling, because the sample might never be recorded while accessing a byte our of the peak. In comparison, the scattered stack usage of **fft** and **qsort** is reflected properly. The different configured temporal granularity only has minor influence on the approximation of these memory regions, thus the overhead can be saved here. However, the results show, that the write count approximation can supply sufficient information about the memory age when the real write distribution has not too much non-uniformity in small

regions. As this non-uniformity in small regions (e.g. a peak of some bytes) anyway cannot be handled by the wear-leveling algorithm later, because relocations are always done at a granularity of 4kB, a proper reflection of small peaks in the approximation would not bring as much benefit as the proper detection of less non-uniform patterns within the distribution. For the **lesolve** benchmark, the accesses to the `data` segment are reflected and can influence the relocation decisions later.

4.3.2 Endurance Leveling

The previous evaluation shows, that the write distribution of an application can be reflected in the approximated write distribution, which is recorded by the software only endurance tracking system. As explained in Section 4.2.2, the example wear-leveling algorithm is integrated into the tracking system and applied on the benchmark applications. Basically, for the tracking system the same configurability is given. The spatial and temporal granularity can be set and thus, the overhead and quality of the endurance tracking can be controlled. For the wear-leveling algorithm, an additional parameter can be configured. The wear-leveling algorithm is triggered from the tracking system, whenever the approximated write count for one memory region reaches a threshold. This threshold is the additional configuration parameter.
As already discussed before, the spatial granularity is set to 4 kB, which is the size of a virtual memory page. The temporal granularity is set to two different values again, a threshold of 1000 writes, called fast balancing, and a threshold of 5000 writes, called slow balancing. The trigger threshold for the wear-leveling algorithm is always set to the minimum of 1, because for the slow balancing configuration this leads to only 7 relocation actions during the entire run of the **bitcount** benchmark. A further reduction of the triggered relocations by changing the parameter might be considerable in case of the memory overhead, but the impact on the quality of the wear-leveling is extremely high, because not every page of the balanced memory can be relocated at least once. This would lead to some pages remaining cold, while the hot pages are only relocated to some other pages.
Figure 4.4 shows the resulting memory trace when the slow balancing configuration is applied on the four benchmark applications. The results can be compared to the baseline without any wear-leveling (Figure 3.8). It can be observed, that the relocation system works and relocates heavy written pages to other pages in such a way, that the entire balanced memory space is used. Specifically for each benchmark application, different observations can be made.

- The **bitcount** application accesses a small part of the memory very intensively, which is detected by the tracking system properly. Thus, the virtual memory page holding this memory region is relocated to the other, less used memory pages. As the

benchmark only causes 7 relocations in total, not every physical page can hold the hot page. This can be fixed by a longer running application or a more frequent balancing. It can be also observed, that the non-uniformity within the virtual memory pages cannot be resolved, because only entire pages are relocated.

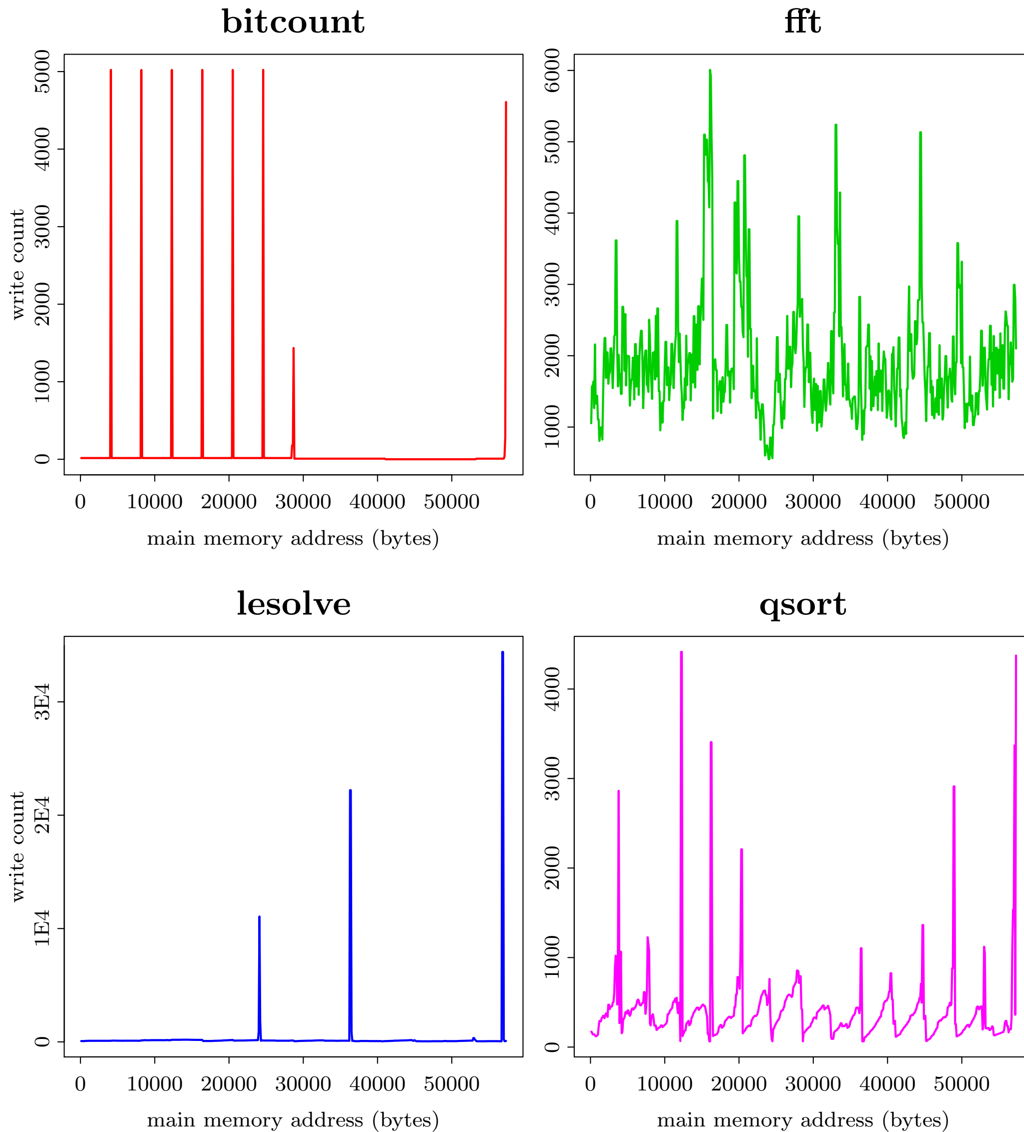

Figure 4.4: Slow Balancing (Endurance Tracking with Integrated Wear-leveling)

- The **fft** benchmark achieves the optically best result from the four benchmark applications. This benchmark uses by default a bigger part of the memory (even more than one virtual memory page) and thus has less non-uniformity within pages. The relocation leads to a more equal write distribution over all the memory in consequence. The approximation of the write distribution provides sufficient information to perform aging-aware relocation decisions.

- The **lesolve** benchmark in contrast achieves a bad balancing, because already the evaluation of the tracking system shows, that the peak write is not captured properly. Thus, the page containing the peak is not relocated to all other physical memory pages. Only the virtual memory pages containing the `data` segment are tracked properly and relocated accordingly.

- The **qsort** benchmark uses again a a bigger memory region, but has higher non-uniformity within virtual memory pages than the **fft** benchmark. Thus, all memory pages show mostly the same write pattern after relocation, which is non-uniform. However, with coarse-grained memory relocations only, an even balance all over the memory cannot be achieved, because the memory regions are always copied to other regions with the same alignment. A non-uniformity within the relocation granularity cannot be resolved. Anyway the results of the **qsort** benchmark show again that relocation decisions are made aging-aware, according to the approximated write distribution.

Setting the endurance tracking system to the fast balancing configuration (write count threshold of 1000 write cycles) has two major effects. On the one hand, the approximated write distribution is captured more precisely and supplies better information to make aging-aware decisions. On the other hand, while keeping the trigger for relocation actions at 1, relocations are triggered 5 times as often as with the slow balancing configuration[6]. This leads to hot pages being relocated around the memory space more often. Different effects of the changed parameter can be observed for the different benchmark applications.

- The **bitcount** benchmark suffers from a low total number of relocations (7) in the slow balancing run. As the number is increased now (36 relocations in total), the hot pages can be relocated to all other memory pages at least once. The uneven distribution of the left and right memory half is caused by the fact, that the simulation is aborted after 36 relocations. Considering the number of virtual memory pages (14), and the fact that relocations target the physical page with the lowest address when pages have the same assumed endurance, the simulation is stopped when the relocation is performed to the the 8th memory page ($36 \mod 14$). This can be identified in the figure.

- The `fft` benchmark shows a more even balance for the fast balancing than for the slow balancing. As the write behavior of the application is not constant during the execution, a higher non-uniformity within the virtual memory pages is achieved while

[6] It is also possible to set the temporal granularity of the endurance tracking system high, but perform not more relocation actions by adjusting the trigger for the wear-leveling algorithm accordingly. As the evaluation of the endurance tracking system already shows, the effects on the approximation quality of the write distribution are only minor, thus the resulting balanced memory trace would not differ much. However, performing an exhaustive analysis of parameter combinations is not in the scope of this thesis.

keeping a hot virtual memory page mapped to the same physical location for a long time. The fast balancing configuration remaps the hot pages more often, thus the write behavior applied to each physical memory pages is shuffled and thus different. This leads to a certainly lower non-uniformity within each page.

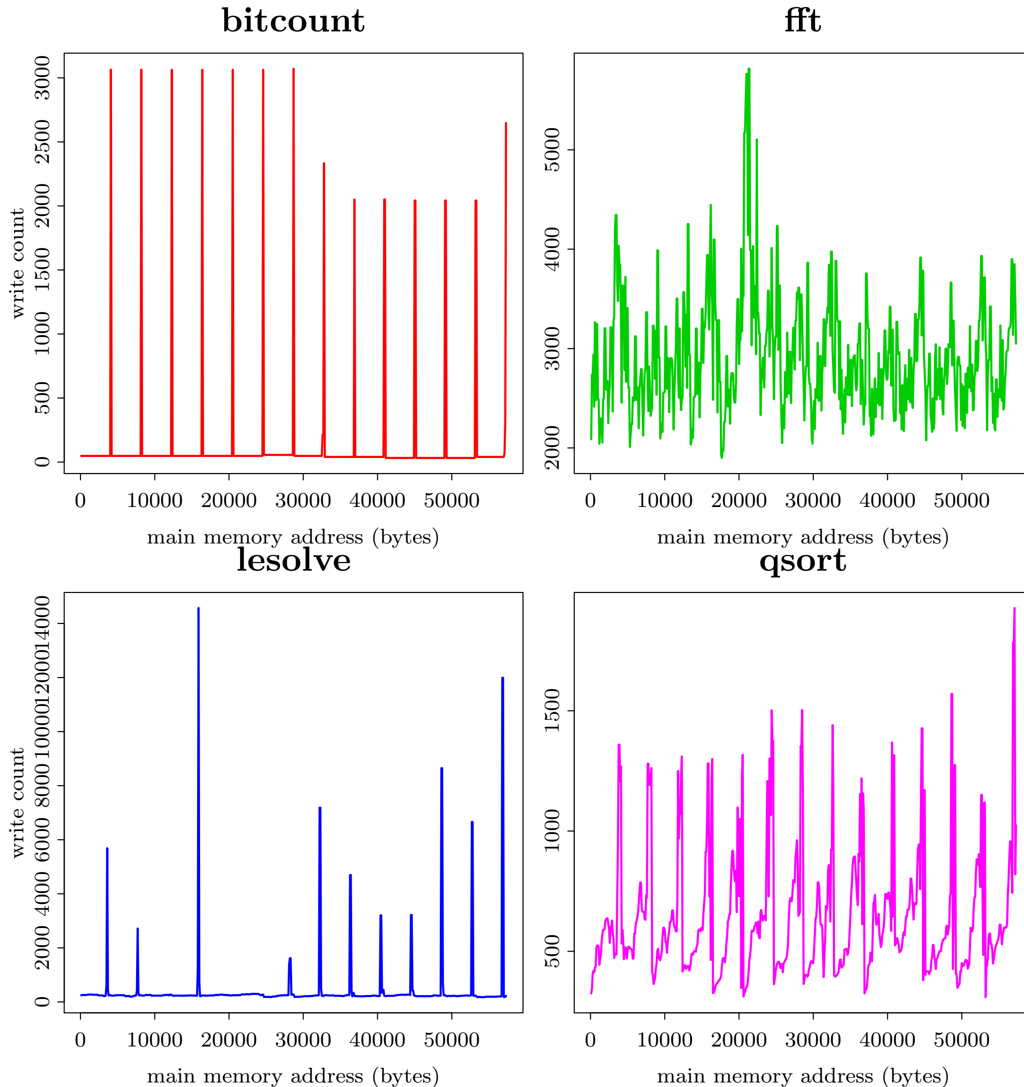

Figure 4.5: Fast Balancing (Endurance Tracking with Integrated Wear-leveling)

- For the **lesolve** benchmark, the increased quality of the approximated write distribution can be observed. The hot page with a small peak of memory accesses is detected better in the approximation and thus, the relocation actions target the hot page more often. Still, the result does not reach an even balance over all the managed memory region, which indicates that the detection of the hot memory region still is not done properly.
- The **qsort** benchmark achieves a better uniformity over all the managed memory with the fast balancing configuration, than with the slow balancing configuration.

Especially the high peak of the last virtual memory page is distributed better over the physical memory region. This is a result of better approximation and more relocation actions. However, the non-uniformity within memory pages could not be resolved much better for the fast balancing than for the slow balancing.

To conclude the evaluation of the wear-leveling algorithm integrated into the endurance tracking system, the achieved endurance according to Equation (1.4) is calculated for the fast and slow balancing configuration.

	bitcount	fft	lesove	qsort
without balancing	0.1%	5.1%	0.2%	3%
slow balancing	1%	26%	0.5%	7.8%
fast balancing	2.7%	43%	2%	29%

Table 4.1: Achieved Endurance of Wear-leveling and Endurance Tracking

Table 4.1 lists the resulting achieved endurance values for both balancing configurations in comparison to the achieved endurance without any wear-leveling actions. The table shows, that for all benchmarks the achieved endurance can be improved. It should be noticed, that the calculation of the achieved endurance is based on the write distribution from the simulation run for the application memory. Thus, the additional writes for the relocation actions are part of the simulated run and influence the calculation. As the additional writes for relocations are completely uniform within pages and are additionally applied according to the age based decisions of the wear-leveling algorithm, they significantly improve the achieved endurance. However, the evaluation of the write overhead shows that for all benchmarks the wear-leveling incurs an overhead of at most 100%. As for all benchmarks under all balancing configurations the achieved endurance is improved by at least a factor of 2, the improvement is not only caused by the additional writes at all.

4.3.3 Overhead

The topic of overhead is already mentioned in the previous subsection. Generally, the endurance tracking system and the wear-leveling algorithm cause two types of overhead:

- Due to additional calculations, management of data structures, interrupt handling, MMU reconfigurations, etc. a certain CPU overhead is introduced. The CPU overhead can be controlled by the frequency of the endurance tracking system and the wear-leveling algorithm. Of course, saving CPU overhead is on cost of approximation and wear-leveling quality in this case.

- Due to relocation actions from the wear-leveling algorithm, a memory write overhead is introduced. As already mentioned, the additional writes are completely uniform

within memory pages (because always an entire page is copied) and are also aging-aware, because the relocation is always performed aging-aware. Thus, the additional writes improve the achieved endurance because the total memory write distribution is more uniform with the additional writes. Concluding from this, the additional memory write overhead should be considered when evaluating the improvement of wear-leveling algorithms (see Section 4.4).

Both types of overhead can be quantified simply with the results of the simulation execution. The required execution time of a simulation can be easily measured in clock cycles, thus the CPU overhead can be determined by comparing the required clock cycles from a baseline simulation and a simulation with endurance tracking (and wear-leveling). The calculation is provided in Equation (4.1).

$$\text{CPU Overhead} = \left(\frac{\#\text{clock_cycles}}{\#\text{clock_cycles_baseline}} - 1\right) \cdot 100\% \tag{4.1}$$

The additional memory write overhead can be obtained similarly by comparing the total number of writes to the memory in the trace files of the simulation. The calculation for the write overhead is provided in Equation (4.2).

$$\text{Write Overhead} = \left(\frac{\#\text{memory_writes}}{\#\text{memory_writes_baseline}} - 1\right) \cdot 100\% \tag{4.2}$$

Using the equations stated above, the overheads can be calculated for the endurance tracking and the endurance tracking combined with wear-leveling for the four benchmark applications.

CPU Overhead	bitcount	fft	lesolve	qsort
endurance tracking only (5000 writes threshold)	0.8%	11.1%	1.5%	2.0%
endurance tracking only (1000 writes threshold)	5.1%	61.5%	9.8%	11.3%
slow balancing	15.1%	188.3%	29.2%	32.6%
fast balancing	76.7%	876.8%	144.7%	164.8%
Write Overhead	bitcount	fft	lesolve	qsort
slow balancing	18.4%	20.2%	20.3%	20.0%
fast balancing	95.0%	93.7%	100.7%	101.2%

Table 4.2: CPU and Write Overhead for the Benchmark Applications

Table 4.2 states the result of the overhead calculations. It can be seen, that the CPU overhead highly depends on the running application (depending on the relation between execution time and the number of memory write accesses). The write overhead turns out to be mostly constant for different applications and the same wear-leveling configuration. This is explained by the fact, that additional write operations, mostly caused by relocations, are triggered on a certain threshold of write operations.

However, the overhead evaluations show that the software only wear-leveling setup might cause a big CPU and write overhead. In certain scenarios (e.g. real-time systems with timing constraints) such overheads might be inconsiderable but an improvement in the achieved endurance is desired. As already stated, the overheads can be reduced by configuring the endurance tracking system and the wear-leveling algorithm accordingly, which usually leads to less improvement in the achieved endurance. In fact, the improvement of the achieved endurance does not directly depend on the frequency of relocation actions, as long as the number of relocations is sufficient and the behavior of the running application is somehow constant. If this scenario is given (e.g. a long or even infinite running algorithm), the wear-leveling can be configured to work on a lower frequency and thus cause a lower overhead and still reach a considerable improvement of the achieved endurance. For instance, the `fft` benchmark is extended to run 20 times as long as the previous benchmark run and the slow balancing configuration is applied. The difference is, that the trigger threshold for the wear-leveling algorithm is set to 15 instead of 1. This results in an achieved endurance of 27.3%, a CPU overhead of 32.4% and a write overhead of 1.39%. Comparing this to the fast balancing configuration for the normal benchmark run, the resulting memory lifetime is mostly the same. The fast balancing configuration improves the achieved endurance by a factor 8.4, while introducing a memory write overhead of 93.7%. Thus, during the memory lifespan the benchmark application can perform 4.4 times as much runs as without any endurance tracking or wear-leveling actions. The slow balancing configuration for the long running benchmark improves the achieved endurance by a factor of 5.4 by introducing an write overhead of 1.39%. In consequence, the application can perform 5.2 times as much runs as without any modifications. Summing up, high memory and CPU overheads can be avoided by an appropriate configuration when the executing software is known to result in a similar write pattern over a long time period.

4.3.4 Comparison to Start Gap Wear-leveling [26]

Comparing the evaluation results of the software only wear-leveling approach in this thesis is not directly possible. This thesis does not focus on the wear-leveling algorithm itself, thus a comparison to other aging-aware wear-leveling algorithms is not reasonable. The software only endurance tracking system can hardly be compared to a hardware based approach, because it is completely unclear by now, how this might be realized and influences the software. In fact, this subsection compares the evaluation results to a non aging-aware, hardware based fine-grained wear-leveling approach, namely Start Gap [26]. This approach requires no hardware for memory write counting, because it is not aging-aware. The hardware for relocations seems to be realizable in a simple manner, because the relocation mapping is calculated analytically and does not require big mapping tables. The CPU overhead for the Start Gap approach cannot be estimated because the influence

of the relocation hardware on the CPU execution cannot be estimated, but for the memory write overhead only the copy actions for relocating memory regions can be assumed to cause an overhead. These actions can be estimated precisely.

The idea of Start Gap is to perform relocations on the granularity of cache-lines. The basic concept is to have a memory region, which is balanced. Additionally to all the cache-lines within this memory region, a spare cache-line, called gap line, exists. The gap line is moved line by line through the balanced memory region, which means the content of the previous cache-line is copied to the gap line and thus, the previous cache-line becomes the new gap line. Once the gap line reaches the upper bound of the balanced memory region, it wraps around to the bottom and continues. This behavior leads to the entire balanced memory region being moved one cache-line further whenever the gap line passes all cache-lines one time. Over time, this leads to a circular movement of the entire memory region and thus, to a non aging-aware balancing of the writes to the underlying physical cache-lines. Due to the fixed remapping scheme from logical to physical cache-lines, the remapping can be done according to a function and not according to big relocation tables. As cache-lines are assumed to be written entirely, no non-uniformity within cache-lines exists and has to be resolved by the wear-leveling scheme. Additionally, this makes the memory write overhead highly controllable by the frequency of gap line movements (Moving the gap line every n th write access leads to a write overhead of $1/n$).

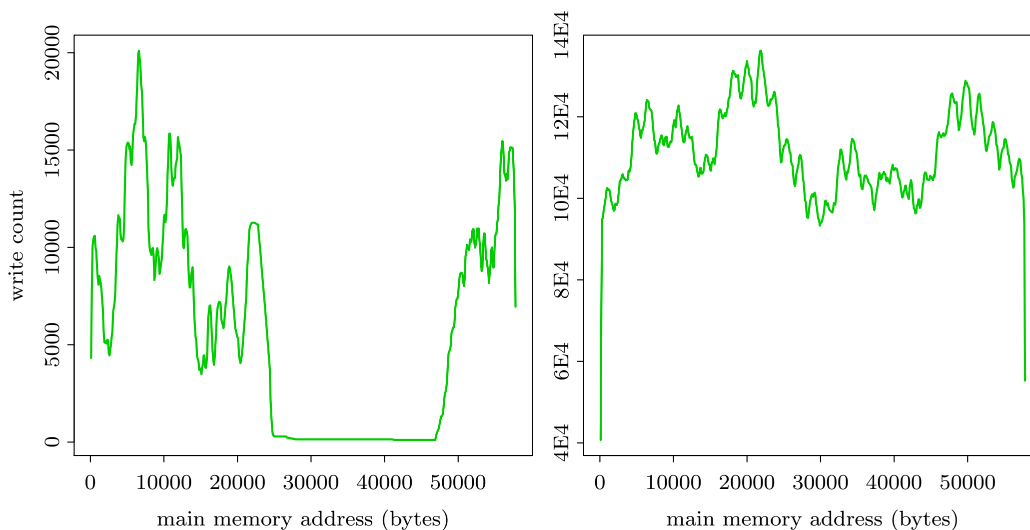

Figure 4.6: Simulated Write Distribution of Start Gap Wear-leveling on the fft Benchmark (Normal (Left) and 20 Times Extended Run (Right))

To compare to the Start Gap approach, a simulator for the resulting memory write distribution is developed and applied. The original trace file from a simulation run is processed and the memory writes are accumulated to the Start Gap remapping function. The cache-line size is assumed as 256 bytes, like in [26]. Cache-lines are always assumed to be written

entirely, thus the write count for all bytes within a cache-line is increased on a write access to a cache-line. The Start Gap algorithm itself is configured to move the gap line every 50 th write access, thus a memory write overhead of 2% is introduced.

In comparison, the software only wear-leveling algorithm is applied on the same application with a configuration, which causes $< 2\%$ write overhead (write count trigger of 5000 for the endurance tracking system and a trigger of 15 for the wear-leveling algorithm). The resulting achieved endurance is presented in Table 4.3.

	normal `fft` run	20x extended `fft` run
Start Gap	27.5%	81.3%
Software Only ($< 2\%$ write overhead)	7.67%	27.73%

Table 4.3: Achieved Endurance of Start Gap Wear-leveling Compared to Software Only Wear-leveling

Figure 4.6 shows, that during the normal benchmark execution not the entire memory region could be targeted by Start Gap approach. This is resolved by configuring Start Gap to perform the relocations faster or by extending the benchmark to run longer. Table 4.3 points out, that for sufficient execution time the Start Gap configuration can achieve a very high improvement of the achieved endurance. Nevertheless, while Start Gap improves the achieved endurance by a factor of up to 15.9, the software only wear-leveling achieves an improvement of a factor of up to 5.4 while introducing the same memory write overhead. Considering that Start Gap operates on a finer granularity, which has no non-uniformity within the relocated memory regions the software only approach is still comparable.

4.4 Discussion

This chapter presents a software only endurance tracking and wear-leveling approach for the main memory, which is independent of the underlying memory hardware. Summing up, the techniques are implemented in a runtime environment and are proven to work. Detailed evaluations on the quality of the approach and the introduced overheads are provided. This section summarizes the key details and points out the insights. First, Section 4.4.1 sums up the technique itself and the observations made during the evaluation. After this, Section 4.4.2 discusses the hardware requirements, which still exist for the software only approach. In the end, the evaluated endurance improvement is related to memory lifetime and the general usability of the software only approach is discussed in Section 4.4.3 and Section 4.4.4.

4.4.1 Summary of Page Based Memory Relocation

Summing up, the key component of the entire system is the memory write distribution approximation. The approximation is recorded by sampling the real write distribution and directly passed to an arbitrary aging-aware wear-leveling algorithm. This enables aging-aware wear-leveling without having access to write counts of the memory hardware. Of course, the approximation is less accurate than hardware provided information, but evaluations show that the approximation is still accurate enough to make aging-aware decisions. The software only endurance tracking system introduces a certain CPU overhead to record the approximated distribution. The overhead can be controlled by setting the sampling frequency of the endurance tracking system, but the evaluations also show, that very uneven write distributions are reflected too imprecisely when the sampling frequency is low.

The wear-leveling algorithm can be chosen to be an arbitrary aging-aware algorithm. For this thesis, a self written algorithm is used, which sorts memory regions according to their (approximated) age in an efficient data structure. Relocations of heavy written page always target the currently least written page. As the only way to perform relocations without the application's corporation is through the virtual memory mapping of the MMU, the granularity of the software only approach is bound to 4kB.

Finally, the evaluation shows a mixed result. It can be concluded, that aging-aware wear-leveling can be done without the support from special hardware and an improvement of the achieved memory endurance can be reached. The introduced overhead by the entire wear-leveling system can also be controlled and kept low for a certain, reasonable setup. However, the evaluations show clearly that the granularity of 4kB is a problem for a lot of applications. The write behavior of the applications is highly non-uniform within the granularity of 4kB, which is mostly caused by the stack region of the executed applications. The non-uniformity within memory pages cannot be resolved by relocating memory pages to 4kB aligned memory regions. It is shown, that an approach, which has no non-uniformity within the relocated memory regions can improve the achieved endurance a lot more. The compared approach in this thesis is a hardware based approach and requires special hardware support.

4.4.2 Summary of Hardware Requirements

Even if the wear-leveling system of this thesis is always called software only, it requires a certain hardware support to work. In comparison to other, hardware based approaches, this approach does not require special hardware, which is not available in common computer systems. The required hardware support for the software only approach can be summarized in a short list:

- A performance counting mechanism with overflow interrupts and programmable counter registers is required to sample the real write distribution. The performance counting mechanism has to support an event, which is directly related to the number of write requests to the main memory (e.g. number of requests on the bus).

- A memory access permission system which allows write accesses to be forbidden and causing a handled trap. The trap handler has to be capable of identifying the memory address, causing the fault. This is usually provided by the access permission system of the MMU.

- A hardware supported mechanism to remap logical memory addresses to physical memory addresses without requiring any corporation from the running application. Usually, the MMU provides this as the virtual memory management.

Obviously the list can be extended by further details (e.g. for the performance counter interrupt also an interrupt controller is required, which allows the software to manage the interrupt), but usually these techniques are widely available in modern CPUs (e.g. Intel or ARM). A lot of microcontrollers do not provide the required hardware support, thus this approach is mainly targeted to full featured CPUs.

4.4.3 Considerations About Memory Lifetime

The evaluations in Section 4.3 calculate the achieved endurance for certain benchmark applications, when different configurations of the wear-leveling system are used. The metric of achieved endurance is always relative to the ideal memory lifetime, the program could reach under a perfect write distribution. Thus, additional overheads caused by the wear-leveling system influence the achieved endurance. For instance, if the wear-leveling system improves the achieved endurance by a factor of 2, this does not mean that the software can perform the double number of computations / actions, etc. before the memory is worn out. The improvement of the achieved endurance indeed says, that the memory write distribution is (measured with the achieved endurance) twice as uniform as before. This means, in mean the improved version can perform the double number of writes before the memory is worn out. Due to the memory write overhead of the wear-leveling system, the application requires more memory writes to perform the computations / actions / etc. in mean. This is measured by the write overhead. However, to identify the improvement of executable computations / actions / etc. before the memory is worn out, the write overhead has to be taken into consideration. Assume the improvement of achieved endurance is given by Equation (4.3).

$$\text{improvement} = \frac{\text{achieved_endurance_with_wear_leveling}}{\text{achieved_endurance_baseline}} \tag{4.3}$$

Further assume the write overhead as a percentage of the baseline execution is given by WO. The real improvement of executable application actions is given by Equation (4.4).

$$RI = \frac{\text{improvement}}{(WO/100) + 1} \tag{4.4}$$

The real improvement can be calculated for the performed benchmarks and is given in Table 4.4. Please note, that this only is related to the number of executable actions before the memory is worn out. The improvement is completely unrelated to required CPU time.

	bitcount	fft	lesolve	qort
slow balancing	8.4	4.2	2.0	2.2
fast balancing	13.8	4.4	5.0	4.8

Table 4.4: Real Improvement (RI) of the Benchmark Executions

The results in Table 4.4 still show that the executable application actions are improved by the wear-leveling techniques. The **bitcount**, **lesolve** and **qsort** benchmarks also benefit form the fast balancing configuration.

4.4.4 Consideratons for Using a Software Only Approach

The comparison of the software-only approach and a hardware-based approach show, that the hardware approach is capable of achieving a significantly better improvement of the endurance than the software only approach. Additionally, some CPU overhead might be saved by applying hardware based techniques, instead of software only techniques. Nevertheless, the software only approach turns out to work and also improve the achieved endurance by a reasonable factor. However, there are important reasons why a software only solution might be favored over a hardware based solution, even if the result is worse:

- For commercial off-the-shelf hardware the hardware based techniques might simply be not applicable at all. The memory controller may not provide an interface to read memory write counts or aging values at all. In this case, the software only solution can still be applied and improve the achieved endurance.
- For customized systems, always the trade-off between functionality and required chip space has to be considered. Memory hardware providing the support for several hardware based approaches definitely requires more chip space than simple memory hardware. With the software only wear-leveling system, the chip space can be saved and used for other purposes. In some scenarios it might be reasonable not to reach perfect endurance and take the additional overhead of the software only solution.

5 Stack Region Write Access Leveling

Previously, Chapter 4 introduces a general purpose approach for aging-aware wear-leveling, only requiring commonly hardware support. This approach does not require the running application to corporate with the wear-leveling service, because the performed actions are totally transparent to the running application. Unfortunately, the limitation to common available hardware limits the performed relocation actions to remappings, using the MMU. Thus, the minimal granularity is the virtual page size, which is 4kB in the experimental setup. Due to this coarse granularity, a highly non-uniform memory usage of the running application within virtual memory pages cannot be tackled by the presented approach. Instead, the entire virtual memory pages are relocated to other physical regions and the same non-uniform write pattern is applied by the application. The evaluations in Section 4.3 show that this problem leads to a suboptimal quality of the entire wear-leveling system.

The main purpose of this thesis is to explore main memory wear-leveling techniques, which do not require the introduction of additional hardware. Thus, a finer granularity for memory remapping cannot be achieved. However, to overcome the issue of inter page non-uniformity, two techniques might be considered, which do not require special additional hardware. Both techniques aim to replace the physical location of memory units, which are significantly smaller than a memory page.

- A relocation of small memory regions can be reached, if the relocation mechanism does not have to be transparent to the application. With a certain application corporation, single variables can be replaced to new physical locations, arrays might be relocated with an offset of at least single CPU words and the stack pointer might be set to a different location for each new function call [22]. These techniques can be implemented in two ways: One way is to perform the relocation logic on the application level (e.g. re-allocate variables on the heap from time to time instead of using the same allocation for a long time or allocate function stack frames on the heap), which requires major modifications to the application logic and to the application code. The other way would be to create a basic infrastructure for relocations at the compile time of the application. For instance, variables could be accessed through an additional indirection layer, which is maintained by the operating system or allocations for new stack frames for function calls can be triggered for every function call. This would require a major modification of the compiler and the application is only

executable in an environment, where the operating system provides the according services. However the fine-grained placement of memory contents is implemented, the key aspect is that the memory placement is abstracted from the operating system and is performed inside of the application. The operating system only has to provide an allocator for fine-grained memory requests, which might be implemented aging-aware (e.g. [22, 28, 21]).

- Without the corporation of the application, the operating system can modify the execution environment of the application anyway. Position independent or relocatable code might be relocated to a new physical location during the execution. Accordingly the stack region can be copied and relocated to another memory region. This technique requires two major steps. First, the application has to be paused by the operating system, because relocations cannot be performed during application code execution. Pausing an application usually is done with interrupts and is well supported by a lot of operating systems. Secondly, the application has to be relocatable at all. If the application is available to the operating system as position independent or relocatable code, the `data` segment might be moved while the application is paused and the `text` segment can be reloaded with the new locations. Also the stack can be copied during the pause and the stack pointer can be adjusted accordingly. When the pause is over, the application continues with relocated locations of fine-grained memory regions (e.g. variables). This technique faces two major advantages. First, the relocation does not have to be integrated into the application logic at all. Secondly, position independent or relocatable code is commonly supported by a lot of compilers and mostly does not require recompiling the software at all. All in all, this technique requires the operating system to take action for each module of the executed application (variables, stack, etc.) instead of providing a common, uniform allocator interface.

Both techniques can help to overcome a high non-uniformity on coarse-grained memory regions, when they are applied accordingly. While the first technique (a common aging-aware allocator) has been studied widely in the literature, it faces a significant disadvantage. When the introduction of wear-leveling actions is performed automatically or the application behavior is not well known at all, it is extremely complex to reach an even wear-level all over the memory. For instance consider the example of stack frames being allocated on the heap for every new function call. This can entirely be automated, even without recompiling the application[1]. First of all, this technique leads to a high fragmentation, because the real used amount of memory for the function's stack frame is not known in advance. In mean, the beginning of the allocated stack frames is used more

[1] A function call is usually compiled to a special assembly instruction, such as `call` for x86 or `bl` form arm. Thus, function calls can be determined easily in a compiled binary and replaced with an appropriate call to a relocation routine.

than the rest of the frames accordingly. Furthermore, the usage of the allocated memory still depends highly on the application itself. An application might spend 90% of the execution time in a single function call, which makes the dynamic allocation of stack frames less effective. Finally, the aging of allocated memory regions can be hardly determined without precise hardware information. Chapter 3 shows that applications may only write single bytes on the stack or write large memory regions on the stack intensively. Thus, the write behavior to allocated memory regions can hardly be predicted and leveled over all allocations. Without getting precise age information from the hardware, an allover even wear-level is very hard to achieve and to maintain for further allocations. Because of this reasons, this technique is not considered to solve the inter page non-uniformity in this thesis.

Fine-Grained Wear-leveling due to Stack Relocations

This thesis proposes a technique, which fits into the second category and changes the execution environment of a running application. As already stated, a specialized technique for every module of the application (variables, stack, text, etc.) would have to be implemented to perform an allover fine-grained wear-leveling. Due to the high complexity, this thesis only proposes one technique, targeting the applications stack. The analysis in Chapter 3 shows, that the stack region faces the highest non-uniformity within coarse-grained sections, thus it is important to target the stack region first. Of course, other techniques for the other application modules can be implemented and applied additionally.

This chapter gives a detailed explanation of the developed stack relocation technique, which resolves the high non-uniformity within the application stack. First of all, the technical workflow and the implementation is stated in Section 5.1. As the movement of the stack to another memory region is not done with the support of any address remapping, the logical addresses of variables on the stack change during the movement of the stack. To keep pointers and references to these variables consistent with the relocation, two possible implementations are provided and described in Section 5.2. After this, an additional extension is introduced in Section 5.3, which allows the application to control the stack relocation process in such a way, that the movement of the stack becomes more efficient. As the stack relocation technique only resolves the high non-uniformity within the stack region, still the page based relocation from Chapter 4 has to be applied to reach an allover even wear-level. The required steps to combine these two techniques are explained in Section 5.4. After evaluating the stack relocation technique only and in combination with the page relocation technique in Section 5.5, the chapter concluded with a final discussion in Section 5.6.

5.1 Stack Frame Relocation

As already motivated, this thesis proposes a technique to resolve the non-uniformity of memory write accesses within the stack region of a running application. As aging-awareness is hard to achieve without introducing special hardware for memory write counting, the technique in this thesis is not aging-aware. The main goal of the technique is to make the write accesses to the stack region mostly uniform, thus an aging-aware technique for coarse granularities (e.g. the one proposed in this thesis in Chapter 4) can easily relocate the entire stack region to different physical locations and reach an allover uniform write distribution. The results in Chapter 3 show, that applications mostly use small amounts of the stack memory very intensively, while the rest of the stack memory remains untouched. To tackle this issue, the stack relocation technique has to move the used stack memory through the memory space in fine-grained steps. This ensures, that even if the application only uses some bytes of the stack, the location of these bytes resides in all the available stack memory at a certain time. Thus, the application writes to all the available stack memory, even if it only uses a small set of variables.

The proposed technique in this thesis moves the stack and the content of the stack by copying the memory content of the stack and adjusting the stack pointer register afterwards. The application uses a different memory location for the stack then, but still accesses the same variables relative to the stack pointer. To achieve an even write distribution without an aging-aware approach, the stack has to be moved in a circular manner through the physical memory to achieve mostly the same write count for each memory cell over time. The idea of the circular movement through the physical memory is adopted from the Start Gap wear-leveling approach [26]. In this section the general concept of the circular movement of the stack memory is explained first in Section 5.1.1. The routine for relocating the stack to a new physical location is implemented in two ways. One can be called directly from the application without interaction of the operating system and is explained in Section 5.1.2. The other one is used by the operating system to relocate the application stack while the application is paused by the operating system. This implementation is interrupt driven and described in Section 5.1.3.

5.1.1 Circular Stack Movement

To achieve a circular movement of the stack while copying it to a new physical location, several aspects have to be considered. First of all, the operating system has to keep track of some memory addresses.

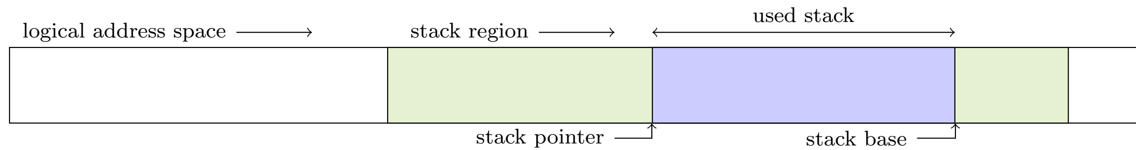

Figure 5.1: Memory Layout of Relocated Stack

Figure 5.1 illustrates the different memory regions and addresses, which are important for the stack relocation. First of all, the operating system has a memory region, which is reserved for the applications stack, which is the green region in the figure. Usually, the application starts using the stack from the highest address of the reserved region. However, with each relocation of the stack, the stack content is copied, thus the beginning of the used stack memory also moves through the memory region. The stack base in the figure illustrates the address, where the first stack content of the application is placed. The stack pointer usually points to the top of the stack, thus everything left from the stack pointer (lower addresses) is unused and everything right from the stack pointer (higher addresses) is currently used stack memory. In consequence, the region between the stack base and the stack pointer contains the current stack memory from the application, which is colored blue in the figure. The application adjusts the stack pointer to the left or to the right, when stack memory is allocated or freed, but never adjusts the stack base. All in all, when the stack is relocated, two major steps are performed:

1. The stack pointer is adjusted to a relocated location (e.g. subtracted 16 bytes[2]). For new variables, the application anyhow uses the relocated stack pointer and place them to the new, relocated position. The access to old variables is also done relative to the stack pointer, thus the adjustment of the stack pointer causes the application to access relocated memory regions for old variables. This requires a second step, namely the copying of the old stack content.

2. As with the relocated stack pointer the application accesses old variables also at relocated memory positions, because the access is made relative to the stack pointer, the old memory content (between stack pointer and stack base) has to be copied according to the relocation of the stack pointer. This also leads to a relocation of the stack base, which is not important at all for the application execution, but for the management information of the operating system.

Repeating these two steps regularly moves new and old stack variables through the memory, reserved for the stack region. Even if the application always writes the same variables,

[2]On ARMv8 architectures the stack pointer has to be aligned to 16 bytes always [1]. Thus the minimum relocation step is 16 bytes.

the adjustment of the stack pointer and the according copying of the content forces the application to access different logical and physical memory locations for the same variables. Performing exactly these steps without any additional setup leads to two significant problems. At first, the relocation will reach the end (lowest address) of the memory region, which is reserved for the stack, at a certain point. As a circular movement is required to achieve an even wear-level, the relocation has to wrap around to the beginning (upper bound) of the stack memory region and continue the relocation. As the access to the stack is always done relative to the stack pointer by the application, a wraparound is not possible without further modifications. Additionally, the application may use arbitrary amounts of memory, beginning from the stack pointer. To satisfy the need of stack memory at every time, the application should be able to always use the same total amount of stack memory, before interfering with other memory regions. For each relocation step, the total available memory for the stack becomes smaller, because the free memory left of the stack pointer shrinks, even if the application does not extend the used stack memory. Both problems make the stack relocation mechanism unusable and are solved in this thesis by establishing a hardware aided wraparound memory region for the memory, which is reserved for the stack.

Hardwrae Aided Wraparound Implementation

The idea of the hardware aided wraparound is to extend the logical address space in such a way, that the available amount of memory left from the stack pointer always is the rest of the memory, reserved for the stack, which is currently unused. Given the total size s of stack memory, which is allocated by the operating system and the size u, which is the amount of currently used stack memory (between stack base and stack pointer), the application has to be able to always use $s - u$ bytes by accessing addresses left of the stack pointer (lower addresses) without interfering with other memory regions. This ensures, that the application can use the full amount of stack memory at any time, by just using the stack in the conventional way.

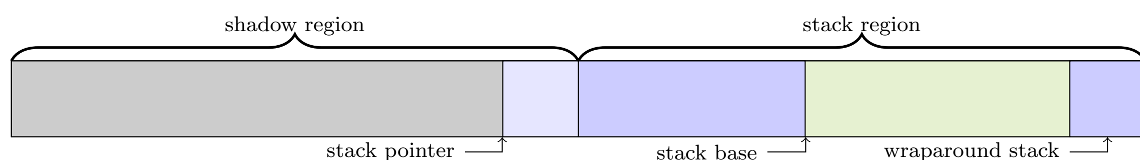

Figure 5.2: Shadow Stack

The hardware wraparound support is implemented by establishing a virtual memory mapping for the stack, which maps the virtual memory pages of the logical stack memory to some allocated physical pages, but also maps the same amount of virtual memory pages

left of the virtual stack memory to exactly the same physical pages. Thus, once the application accesses memory contents left of the reserved stack memory, the accesses are redirected to the physical pages, which are mapped to the upper bound of the reserved stack memory. Figure 5.2 illustrates the setup of the so-called shadow stack. The stack pointer already crossed the border between the real and the shadow stack, thus the stack content which is out of the real stack region is redirected to the upper end of the real stack region. This implementation has several consequences:

- As long as the stack base is located in the real stack region and not in the shadow region, the application can access the full amount of stack memory by accessing addresses left of the stack pointer.
- The wraparound for the stack relocation can be implemented correctly. When a relocation would move the stack base into the shadow region, the stack pointer is not just relocated (subtracted the relocation offset), but additionally put back into the stack region (add the stack memory size). Given an address x in the shadow region, the memory content at $x +$ size of stack memory inside the stack region is exactly the same as the content at x. Thus, setting the stack pointer from the shadow region to to the according location in the stack region does not change the application's perspective on the memory content at all.

All in all, the application might use memory addresses in the shadow region and the operating system relocates the stack later back to the stack region. As both are fully compatible, the application can continue the normal execution in this scenario. This allows the operating system to perform relocations in a circular manner, such that the used stack memory is moved around all the stack memory again and again. During all this process, the application can access the full stack memory at any time by just accessing addresses relative to the stack pointer. The only requirement is a virtual memory region, which has twice the size of the reserved stack memory and an appropriate setup of the virtual memory as the shadow and real stack.

5.1.2 Synchronous Relocation Implementation

Once the memory is configured accordingly to enable the shadow stack, the relocation has to be performed regularly to balance the wear-level of the memory cells of the stack region. The relocation routine itself exists in two variants, one which can be called from the running application to perform a relocation as part of the application execution and another one, which can be performed by the operating system during an interrupt of the application. The version, which is called as part of the application execution has the advantage, that it can be called at a smart point of the execution, when the currently used stack memory is small and thus the overhead for copying the stack is smaller. However,

copying the stack of the application requires the stack to not be touched during the copying. As the synchronous relocation is called from the application, it is executed on the exception level of the application and cannot use another stack, than the applications one. Of course, the synchronous relocation could be implemented as a systemcall, but this would introduce an additional overhead.

To not touch the stack during the synchronous relocation routine, it is entirely written in assembly and only uses caller-saved registers. The implementation basically consists of two steps. At first, the old stack content is copied to the relocated stack addresses (moved left by the relocation offset). After this, the CPU registers handling the stack usage are adjusted. The relocation routine is implemented compatible to the function call ABI, thus it can be called as a usual function from the application. Once this function returns, the stack pointer is relocated relative to the stack pointer, the function was called with.

```
begin_loop:
    ldr x14,[x13]
    str x14,[x13,#-RELOCATION_STEP]
    add x13,x13,#0x8
    cmp x13,x12
    blt begin_loop
```

Listing 5.1: Synchronous Relocation - Stack Copying

Listing 5.1 is an excerpt from the synchronous relocation implementation and shows how the old stack content is copied to the new, relocated location. The spare register `x14` is used to load a memory word and store it to a new location. x13 holds the current address (in the not relocated stack), which is copied. First the memory content is loaded and stored at the relocated position afterwards (old address minus the relocation step). The rest of the code is a simple loop implementation to repeat the code for the entire stack.

```
sub sp,sp, #RELOCATION_STEP
sub x29,x29,#RELOCATION_STEP
```

Listing 5.2: Synchronous Relocation - Register Adjustment

Listing 5.2 shows the seconds step of the relocation, namely the adjustment of the registers. Of course, the stack pointer has to be adjusted regarding the stack relocation, but also the base pointer register x29 has to be relocated accordingly. The base pointer usually holds the stack pointer, a function was called with. As the relocation routine does not touche the base pointer, it contains the stack pointer, the calling function was called with. As the entire stack is moved, also the base pointer has to be adjusted, to point to the correct memory location.

5.1.3 Interrupt Relocation Implementation

As already mentioned, the relocation routine is also implemented in a second version to be executed during an interrupt handling by the operating system. The disadvantage of the

synchronous relocation in Section 5.1.2 is that the application has to call the relocation routine during execution. This requires at least modifications in the source code of the application and a recompilation. If the source code is not available or it is unreasonable to modify the source code, the only chance is to perform the stack relocation outside of the application. To do so, the operating system has to interrupt the application regularly[3] and perform a stack relocation during the handling of the interrupt.
The implementation of the interrupt relocation routine differs slightly from the synchronous relocation routine. First of all, it is written in C++, because the interrupt is executed on the runtime system stack and not on the application stack, thus the application stack can be modified from the C++ code, because it is unused during the interrupt handling. The fact, that the relocation is performed on anther execution level and on another stack also requires a different way of register accesses. As for every interrupt handling, the register set at the time of the interrupt is stored by the interrupt handler first, to not be modified by the interrupt handling functions. These stored registers are written back to the register set, when the interrupt handling finishes. Thus, to modify the registers, like the base pointer register x29, the relocation routine has to modify the backed up register set. The stack pointer itself does not need to be backed up, because the ARMv8 CPU has separate stack pointers for each exception level. The stack pointers from lower privileged exception levels can be modified, by writing the according system register.

```
uint64_t app_sp;
asm volatile("mrs %0, sp_el0" : "=r"(app_sp));
app_sp -= RELOCATION_STEP;
asm volatile("msr sp_el0, %0" ::"r"(app_sp));

uint64_t lword = saved_stack_base[30];
lword -= RELOCATION_STEP;
saved_stack_base[30] = lword;
```

Listing 5.3: Interrupt Relocation - Stack Register Adjustment

Listing 5.3 shows the adjustment of the stack pointer and base pointer register. The stack pointer from the applications exception level (EL0) is accessed as a system register with inline assembler calls. The base pointer (register x29) is adjusted by modifying the backed up register set.

```
for (uint64_t target = app_sp + RELOCATION_STEP; target < __current_stack_base_ptr;
            target += 8) {
```

[3] The most uniform wear-level can be achieved, when the stack relocation is triggered after a number of performed writes to the stack. Thus, under the assumption of a slowly changing write pattern to the stack, every memory region of the stack memory would be written mostly equal. In this thesis, the stack relocation mechanism is implemented in the runtime system, descibed in Chapter 3, thus the support for performance counter overflow interrupts of `BUS_ACCESS_ST` events is already implemented. Performing the stack relocation during the interrupt handling of the performance counter overflow results in a stack relocation after mostly equal often writes to the stack.

```
        uint64_t lword = *((uint64_t *)(target));
        *((uint64_t *)(target - RELOCATION_STEP)) = lword;
}
```

Listing 5.4: Interrupt Relocation - Stack Copying

Listing 5.4 shows the core code of the stack copy loop, which copies the old stack content to the new location. The implementation performs exactly the same computation like the copy routine in the synchronous relocation implementation in Listing 5.1.

5.2 Address Consistency

As the previous section states, the key idea to evenly balance the wear-level inside the stack region is to copy the used stack memory in a circular manner through the stack memory and adjust the stack pointer accordingly. This makes sure, that even if the application always writes to the same variable, the write accesses target different memory addresses over time and thus increase the wear-level of different memory regions, instead of always the same. However, copying the already used stack memory implies a change of the location of currently used local variables. For usual code execution this is not a big problem, because gcc compiled applications usually access local variables relative to the current stack pointer. As the stack pointer is adjusted according to the new locations of the variables, the accesses still target the correct memory content. The changed locations of local variables become a problem, if pointers or references are held to these variables. For instance, a function might prepare some arrays with data on the local stack and pass pointers to these arrays to a processing function. If a stack relocation is performed during the execution of the processing function, the locations of the arrays change. The pointers, which are usual variables on the stack of the processing function are relocated and accessed correctly, but still contain the old addresses of the arrays. An access to these pointers returns invalid results in consequence. This problem is not only bound to the situation, when the programmer actively creates pointers or references in the source code, usually the base pointers are stored on the stack whenever a function is called. This builds a linked list of base pointers, which enables debugging actions like stack frame tracing. Thus, the base pointers are pointers into the stack, which are stored as usual variables on the stack. On a relocation, the address of these variables changes, but the content (i.e. the address to the following base pointer) stays the same. Thus, the pointers point to invalid locations and an access would lead to a wrong behavior.
To overcome this issue, this thesis proposes two different solutions. For both solutions the linked list of base pointers has to be adjusted after every stack relocation, which can be done by simply following the list (respecting the missing offset in the pointers) and adjusting each pointer according to the relocation offset. However, one solution is to adjust all pointers existing on the stack and pointing to variables on the stack accordingly

while performing the stack relocation. This solution is explained in detail in Section 5.2.1. Another option is to not allow the application to use raw pointers at all. Instead, the application can only use a special smart pointer, which checks on each dereferencing how far the stack is relocated and adjusts the memory access accordingly. This solution is described in Section 5.2.2. Of course, both solutions may have to be used at the same time. As the first solution only relocates local pointers, which are stored on the stack, pointers anywhere else cannot be adjusted accordingly. It might happen, that an application creates some local variables and passes pointers into a data structure, which is allocated on the heap. If the stack is relocated while the pointers on the heap are valid and in use, they are not adjusted, even if pointers on the stack are adjusted. This requires to not pass raw pointers to data structures out of the stack at all. If the smart pointer solution is used, the pointer is checked and adjusted, even if it resides on the heap, because the smart pointer checks the current stack relocation irregardless of the position of the smart pointer.

5.2.1 Raw Pointer Adjustment

The first solution to overcome the problem of invalid pointers to the stack after stack relocations in this thesis is to adjust pointers, which reside on the stack, during the relocation process accordingly. Basically, variables on the stack are local variables and only temporary available. Thus, pointers to these local variables should also only exist temporary as local variables on the stack. However, as already explained there might be reasons why pointers to local variables may also be passed to data structures outside of the stack, but this scenario is not covered by this solution. The implementation can be easily done as an extension to the already presented relocation routines. The used stack content is copied word by word to a new location and during this, any word of the stack content is loaded into a register. If the word is identified as a pointer to a stack variable, the value is adjusted according to the relocation offset of the stack. Unfortunately identifying a memory word on the stack as a pointer is a complex task. From the perspective of the relocation routine, every word on the stack is just a 64bit number without any special meaning. A program variable containing the value 43981 is exactly the same memory word as a pointer to the address $0xABCD$. Even for a compiler it would be hard to determine which word on the stack is a pointer or could be used as a pointer in future, because C / C++ allows arbitrary castings from any data type into a pointer. To solve this problems, the solution in this thesis puts a strong limitation to the memory usage of the application. For local variables, the application is only allowed to use at most 32bit datatypes, but they have to be stored with 64bit alignment[4].

[4]The GCC compiler supports aligned placement of variables easily. Even if this is not possible, the application can use 64bit datatypes and has to make sure to only write values, which only set the 32 lower bits.

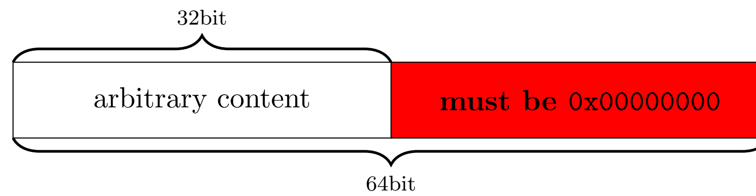

Figure 5.3: 64bit Aligned Data Types

This makes sure, that every 64bit word in the stack memory, which corresponds to any variable of the application contains a value x with $0 \leq x \leq 0xFFFFFFFF$. The virtual memory pages for the stack memory are placed at a location, which is out of the 32 bit address space, thus any address of the stack has a bigger value than $0xFFFFFFFF$. In consequence, every pointer is represented on the stack as a memory word, which is bigger than $0xFFFFFFFF$. As all other memory contents from the application are forced to have values lower than this, pointers can be identified just by the value of the memory word.

```
for (uint64_t target = app_sp + RELOCATION_STEP; target < __current_stack_base_ptr;
            target += 8) {
        uint64_t lword = *((uint64_t *)(target));
        if (lword < __current_stack_base_ptr && lword > __shadow_stack_begin) {
            lword -= RELOCATION_STEP;
        }
        *((uint64_t *)(target - RELOCATION_STEP)) = lword;
}
```

Listing 5.5: Raw Pointer Adjustment in C++

Listing 5.5 shows the C++ version of the stack content copy loop with the raw pointer adjustment implementation. Whenever the copied memory word contains a value, which is a pointer to the stack, it is also adjusted to match the new location of the stack. Of course, also the register set has to be checked and adjusted, because a register might currently contain a pointer to a variable on the stack. This does not have to be done only for the interrupt relocation routine, but also for the synchronous relocation routine. A pointer to a variable on the stack might be in a callee saved register, thus these registers have to be checked and adjusted by the synchronous relocation routine.

All in all, the test applications for this thesis are implemented according to the requirement of 64bit aligned datatypes with values less than 32bit and the functionality is maintained. The relocation of variables on the stack and adjustment of pointers do not change the application behavior or results.

5.2.2 Smart Pointer Adjustment

The second solution introduces a smart pointer, which could be used instead of a raw pointer, whenever a pointer to a local data structure is created and passed somewhere. The smart pointer is able to execute additional source code while dereferencing the pointer. This source code checks the current relocation of the stack and adjusts the pointer accordingly. If the application only uses the smart pointer for local variables and data structures, the raw pointer adjustment (Section 5.2.1) may be disabled. However, the linked list of base pointers still has to be adjusted on each stack relocation. Additionally, the raw pointer adjustment and stack pointer adjustment can be combined and used at the same time. For instance when pointers to local variables are passed to a heap data structure, the smart pointer has to be used to still maintain correctness. For usage on the stack (no passing to heap data structures), raw pointers may be used at the same time and handled by the raw pointer adjustment.

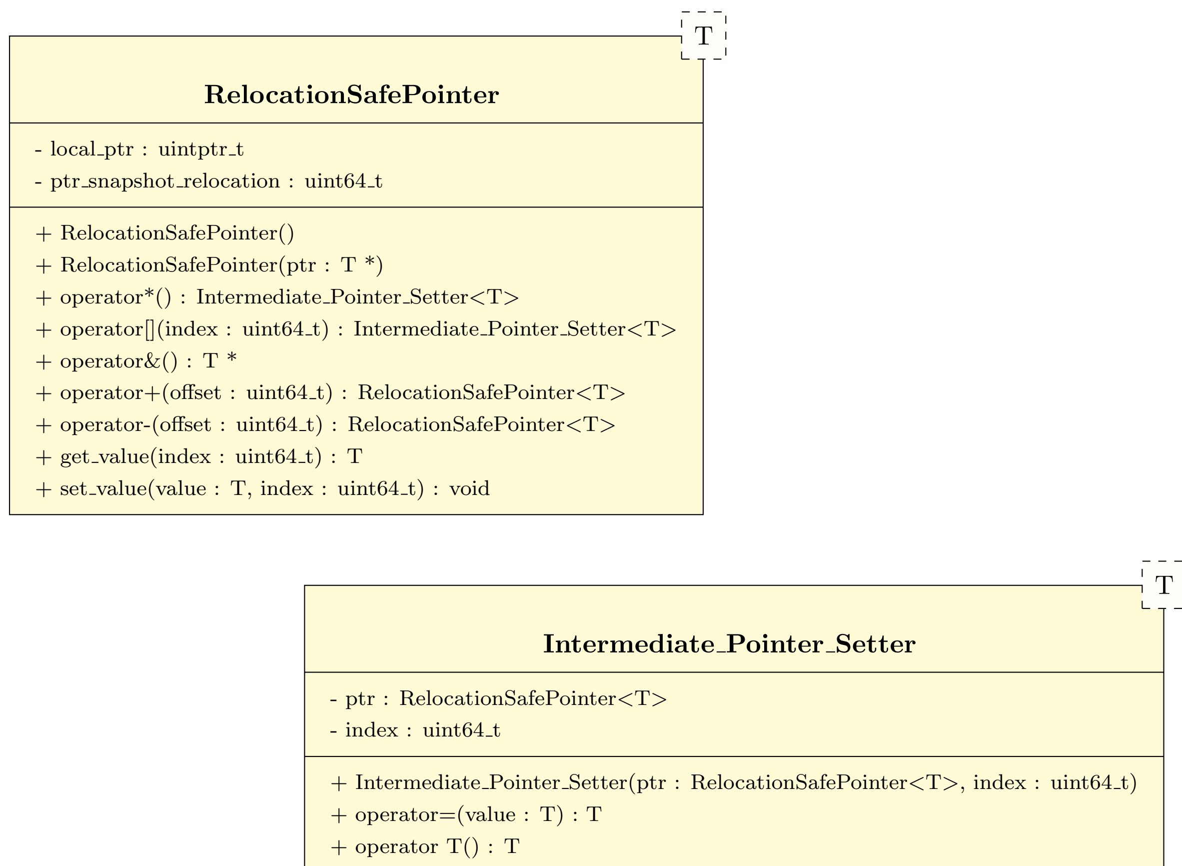

Figure 5.4: Smart Pointer Class

Figure 5.4 lists the detailed interface of the smart pointer implementation. Basically, the class RelocationSafePointer holds a raw pointer to the object, pointing to. In addition to the raw pointer, a snapshot of the current stack relocation offset is kept (the difference

between stack base and the upper end of the allover stack memory region). Using these two variables during the dereferencing, the current stack relocation offset can be determined and the required raw pointer adjustment can be calculated. The initial assignment of the variables is done during the initialization constructor, which gets a raw pointer as an argument. If the raw pointer adjustment is not activated, or the smart pointer is not stored inside the stack, the passed raw pointer parameter might become invalid because of a stack relocation. The only possibility to overcome this is to force stack relocations to pause while calling the constructor. This is provided as a runtime system service in the implementation. Listing 5.6 is an excerpt from the implementation of the constructor. The stack relocation snapshot is stored just as already described.

```
local_ptr = (uintptr_t)ptr;
local_ptr |= (0b1UL << 63);
ptr_snapshot_relocation = (__virtual_stack_end - __current_stack_base_ptr);
```

Listing 5.6: Smart Pointer Construction

Storing the raw pointer requires an additional step. The smart pointer may reside in the stack and raw pointer adjustment may be enabled. This would cause the internal raw pointer of the smart pointer being adjusted. As the adjustment is already done during the dereferencing, this would lead to an incorrect result. To protect against this behavior, the upper bit of the raw pointer is set to 1, thus the raw pointer adjustment does not recognize the corresponding memory word as a pointer to the stack and does not adjust it. For dereferencing, the upper bit has to be simply reset to 0. As an optimization, the smart pointer checks if the raw pointer is a pointer to the stack and only performs the previously described steps in this case. For raw pointers to other regions no special actions are required.

```
T get_value(uint64_t index) {
    if (local_ptr & (0b1UL << 63)) {
        syscall_delay_relocation();
        uintptr_t correct = local_ptr & ~(0b1UL << 63);
        uint64_t current_relocation =
            (__virtual_stack_end - __current_stack_base_ptr);
        if (current_relocation >= ptr_snapshot_relocation) {
            correct -= (current_relocation - ptr_snapshot_relocation);
        } else {
            correct += (ptr_snapshot_relocation - current_relocation);
        }
        T copy = ((T *)(correct))[index];
        syscall_continue_relocatuion();
        return copy;
    } else {
        return ((T *)(local_ptr))[index];
    }
}
```

Listing 5.7: Smart Pointer Dereferencing

For the dereferencing of the smart pointer, again it is checked first, if the pointer is a pointer to the stack. This can be determined easily by the set most significant bit, which protects pointers into the stack from the raw pointer relocation. If the pointer is not a pointer to the stack, it is just dereferenced and the value is returned, which can be seen in Listing 5.7. For the case of a pointer to the stack, stack relocations have to be stopped[5] during the dereferencing, because this could make the temporary calculated, adjusted raw pointer invalid. After this, a correct raw pointer is calculated by setting the most significant bit of the internal raw pointer to 0 and comparing the snapshot stack relocation with the current stack relocation. The case that the stack relocation wrapped around and the real content now is ahead of the stored internal raw pointer is respected properly. The function `get_value` itself is used to access an array element beginning at the raw pointer location. This can be also used by non array pointers by just setting the `index` to 0. The implementation for writing new values into the smart pointer just differs in the final raw memory access. The rest of the smart pointer implementation is just a syntactical enhancement, to allow the same syntax on the smart pointer as on raw pointers.

To simplify the need of the correct smart pointer creation, an allocation macro for stack memory is also provided in the implementation (Listing 5.8).

```
#define stack_allocate(type, size, name)            \
    syscall_delay_relocation();                     \
    char __auto_alloc_##name[sizeof(type) * size]; \
    RelocationSafePointer<type> name((type *)__auto_alloc_##name);
```

Listing 5.8: Stack Memory Allocation

This macro disables stack relocations first, allocates the requested amount of memory for a data array on the stack and creates a proper smart pointer instance in the end. The constructor of the smart pointer continues stack relocations by performing the according system call in the end. The allocation has to be used instead of creating local arrays or variables and creating pointers to these.

```
stack_allocate(uint64_t, 5, test_arr);
test_arr[0] = 51;
test_arr[1] = 52;
test_arr[2] = 53;
test_arr[3] = 54;
test_arr[4] = 55;

stack_allocate(uint64_t, 1, t1);
*t1 = 42;
stack_allocate(uint64_t, 1, t2);
*t2 = 182600;
```

Listing 5.9: Smart Pointer Usage Example

[5] In fact, the implementation only delays stack relocations. This means that requests to relocate (e.g. from an interrupt driven mechanism) are noticed and executed when the continue systemcall is performed.

Listing 5.9 gives an example how the stack allocation and smart pointer access can be used. Notice that no raw memory access is performed by the code, the syntax calls the appropriate functions of the smart pointer. Passing of the smart pointer to other functions or data structures can be simply done by call by value copies of the smart pointer. The automatic generated copy constructor just moves the two member variables to a new instance of the smart pointer, which does not have to be secured because pointers to the stack are already secured by the most significant bit. Note that in case of activated raw pointer adjustment the constraints for memory usage on the stack (maximum 32bit used in 64bit aligned variables) still have to be respected.

5.3 Synchronous Hinting Environment

The previous sections state in detail the concept of the circular stack frame relocation and how to maintain the correctness of pointers, even with relocated variables. Another quiet important aspect is the performance of the circular stack frame relocation. As for every relocation step the currently used stack memory has to be copied to the new location, a certain overhead regarding the number of memory writes and the execution time of the relocation mechanism is introduced. Of course the overhead can be controlled by the frequency of the relocations, but this also influences the resulting wear-leveling quality. Another way to control the overhead up to a certain degree is the time when the relocation of the stack is performed. Applications usually don't use the same amount of stack memory over time. For instance, local variables, arrays, etc. which are created and used inside a loop are allocated on the stack at the beginning of the loop body and deallocated at the end of the loop body. Thus, relocating the stack at the beginning of the loop body, before the variables are allocated, would be a good decision and save some overhead because the amount of stack memory to be copied is smaller. Additionally, for recursive algorithms it would be a good decision to perform the relocation of the stack when the calling depth is low, respectively the number of function frames on the stack is low.

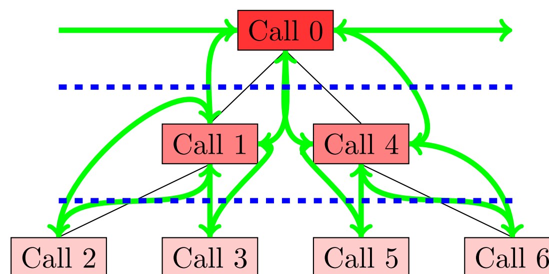

Figure 5.5: Call Graph for a Binary Recursive Algorithm

Figure 5.5 illustrates how a binary recursive algorithm (every function performs two recursive calls) is called. The dashed blue lines separate the levels of calling depth. To reach a

total number of relocations during the execution (e.g. 2 or 4 in the figure), the best choice would be to perform the relocation on every subroutine call of Call 0 or Call 1 and Call 4. To achieve a lower overhead, the provided implementation of the relocation mechanism allows the application to give hints to the runtime system. This means, the application can give a hint, whenever a relocation would introduce less overhead (e.g. because at the beginning of the loop the variables are not allocated) and the runtime system decides if a relocation is performed at the time of the hint or not. This decision is mainly based on the configured frequency for stack relocations, which has to be maintained. Section 5.3.1 first gives a brief overview how the hints are processed from the runtime system. After this, Section 5.3.2 and Section 5.3.3 present two self adaptive implementations, which aim to reduce the number of unnecessary hints automatically.

5.3.1 Hinting Decision

The hints from the application are given to the runtime system by calling a simple function. The implementation can be seen in Listing 5.10. Basically, the runtime system maintains a variable, namely `performed_since_last_irq`, which is set to false on every interrupt, which aims to relocate the stack. If the interrupt handler finds the variable set to one, it does not perform a relocation during the interrupt, because it was already performed by a hint before. If the hint function finds the variable set to true, it will also not perform a synchronous relocation[6], because no relocation was requested from an interrupt since the last successful hint, thus the frequency of hints is higher than the configured frequency for relocations.

```
void StackBalancer::hint_relocation() {
    if (!performed_since_last_irq) {
        performing_balance = true;
        performed_since_last_irq = true;
        performed_irq_version = false;
        relocate_stack();
        relocation_count_syn++;
        performing_balance = false;
    }
}
```

Listing 5.10: Stack Relocation Hinting Interface

The hinted relocations and the relocations performed during an interrupt are synchronized weakly. It has to be ensured, that not two relocations will happen at the same time, thus the hint function sets a flag, that it currently performs a relocation. When the interrupt handler finds this flag set, it does not perform a relocation. This does not prevent an interrupt performing a relocation between the checking of `perfomed_since_last_irq` an

[6]The hint function is called directly from the application, without performing a system call. Thus, only the synchronous version for relocations can be used.

the setting of the flag. Usually an interrupt should not perform a relocation in this situation, because the `perfomed_since_last_irq` is set to true after the successful hint. However, this does not affect the correctness of the relocation at all and a complete synchronization, e.g. interrupt synchronization, may introduce much more overhead.

As already stated, the hints have to be given by the application, as only the programmer knows when the currently used amount of stack is small. This may lead to a too big number of hints, because a hint may be given at every beginning of a loop iteration. For a low configured frequency of relocations, this would make the most hints unsuccessful and useless. Because of this, the implementation in this thesis also provides two self adaptive mechanisms, which adjust the number of hints automatically to the configured frequency of relocations.

5.3.2 Self Adaptive Loop Hinting

The idea of the self adaptive loop hinting is to give hints at the beginning of the loop body, when the amount of used stack is small, but to give hints not on every loop iteration. The threshold, after how many loop iterations a new hint is given, is determined similar to a closed loop control system. Whenever an interrupt relocation was necessary since the last hint, the threshold is lowered and more hints are given. If a hint was unsuccessful, because not relocation was required, the threshold is increased and less hints are given.

```
void StackBalancer::outer_loop_automatic() {
    if (outer_loop_cycle++ >= outer_loop_balancing_ratio) {
        if (performed_irq_version) {
            if (outer_loop_balancing_ratio > 1) {
                outer_loop_balancing_ratio--;
            }
        }
        if (performed_since_last_irq) {
            outer_loop_balancing_ratio *= 2;
        }
        hint_relocation();
        outer_loop_cycle = 0;
    }
}
```

Listing 5.11: Stack Relocation Adaptive Loop Hinting

Listing 5.11 shows the function, which can be called from the application at the beginning of a loop instead of the hint function. The threshold for performing a hint is checked and adjusted according to the the captured information (`performed_irq_version` indicates the last relocation was made from the interrupt handler and not from the hint, `performed_since_last_irq` indicates another hint already was successful and no relocation is needed). Note that this function should not be called in nested loops, because the hinting threshold is a global variable.

5.3.3 Self Adaptive Recursion Hinting

The concept of the self adaptive recursion hinting is very similar to the self adaptive loop hinting in Section 5.3.2. The difference is, that not the loop iteration is used to determine if hint is given, instead the recursion depth is used. To determine the recursion depth in an automatic way, the application only has to put the macro STACK_RECURSIVE_FUNC into the main recursive function. This macro creates an empty object on the stack, thus the constructor is called immediately and the destructor is called at the end of the function.

```
#define STACK_RECURSIVE_FUNC RecursiveGuard __r__;
RecursiveGuard::RecursiveGuard() {
    StackBalancer::instance.recursive_automatic_call_begin();
}

RecursiveGuard::~RecursiveGuard() {
    StackBalancer::instance.recursive_automatic_call_end();
}
```

Listing 5.12: Stack Relocation Adaptive Recursive Hinting

These two hooks allow to count the allover current recursion depth. The rest of the implementation is very similar to the self adaptive loop hinting. On a certain threshold of the recursion depth a hint is performed. If the number of hints is too low, the recursion depth trigger is increased and more hints are performed. If too much hints are performed, the recursion depth trigger is decreased and less hints are performed.
Summing up, both self adaptive techniques required the application to only put a single line at the beginning of the most outer loop, respectively the beginning of the main recursive function. Doing this, the adaption aims to perform the most stack relocations as successful hinted relocations. This may help to reduce the overhead for stack relocations by performing the relocation at the best point of the application execution.

5.4 Combination With Page Based Relocation

The stack wear-leveling approach, presented in this chapter, aims to make the write count to memory regions within the stack memory more uniform. Thus, only the stack region of the memory is touched. The stack region typically is only one memory region and composes together with the text, data, bss and heap region the memory image of an application. Because of this, the spatial allocation of the stack memory is only one part of the total memory image, which might be small compared to the data or heap segment for instance. Making the wear-level uniform only within this small memory region might achieve a non optimal allover wear-level, because the text segment for instance is usually not written at all. Thus, in addition to the balanced stack region it is also important, to move heavy written (hot) memory regions around the memory and place it to less written (cold) physical memory locations. Such a scheme is presented in detail in Chapter 4 and

is based on a relocation of physical memory pages mapped to virtual memory pages. It is already stated, that the stack wear-leveling approach is meant to be as an extension to the page based, coarse-grained relocation approach to overcome the shortcomings of this approach. This section states shortly how both approaches are combined and executed together.

Implementation Details of the Integration

Generally, the page based relocation and the stack region write access leveling can run together with no problems. The stack region leveling can be interpreted as a part of the application (as it can be executed fully as part of the application) and the page based relocation as a runtime system service, which does not change the application's perspective at all. The page relocations are performed during interrupt handling routines, which completely pause the running application. After the relocation finishes, the perspective of the application is the same as before, thus even a stack relocation might be interrupted by a page relocation without any problem. In addition to the stack relocations triggered directly from the application, the runtime system also implements the stack relocations as an interrupt driven mechanism, which acts completely without the corporation of the application. This mechanism is configured with a frequency for stack relocations, which relates to the total number of writes to the memory, for instance a stack relocation should be performed every 5000th memory write. Luckily, the page based relocation already implements an interrupt driven mechanism, which causes an interrupt after a configurable number of writes to the allover memory. This mechanism is used for the statistical write distribution approximation, as well as for the triggering of page relocations. As long as as the configurations for all mechanisms are compatible (i.e. the frequencies are multiples of the base frequency of one mechanism and can share the memory write count interrupt mechanism) only one interrupt driven memory write count mechanism is required. The interrupt handler of this mechanism calls the write count approximation, the page balancing and the stack balancing strongly in order, thus the executions do not overlap and influence each other.

Finally, the implementation of the write count approximation and the page balancing have to be fully aware of the stack balancing, because they are based on modifications to the virtual memory mapping. The stack balancing mechanism establishes a specialized virtual memory configuration due to the shadow stack (Section 5.1.1), which assigns two virtual memory pages to each physical page of the stack memory. As the write count approximation and the page balancing both operate basically on physical pages, they have to aggregate the results from both virtual memory pages, respectively maintain the shadow stack mapping when relocating physical pages. This also points out, that the stack relocation cannot be implemented entirely from the user space (e.g. in a standard Linux

application). The mechanism always requires the operting system to setup at least the required static configuration. However, the write count approximation and page relocation implementations basically exist in two variants, one operating on a stack memory page and another one operating on a usual memory page. A global configuration option enables or disables stack balancing at all, which also changes the behavior of the write count approximation and the page balancing. As the physical memory pages for the stack do not change at all with enabled stack relocation, only the logic of the write count approximation and page balancing has to be adjusted. The wear-leveling decisions to the physical pages remain completely unchanged.

5.5 Evaluation

The stack region write access leveling presented in this chapter is designed as an extension to the page based relocation (Chapter 4), to overcome the shortcomings of this approach. Nevertheless, the stack region write access leveling can be applied without the page based relocation to only make the wear-levels inside the stack more uniform. This section intends to evaluate the stack region write access leveling in detail. To do so, Section 5.5.1 first gives a short overview about the executed benchmarks and how they are prepared to match the address consistency requirements of the stack relocation. After this, the stack region write access leveling is evaluated regarding the resulting write distribution in Section 5.5.2. The combination with the page based relocation is evaluated subsequently in Section 5.5.3. Finally, the introduced overheads and effects of the overheads are discussed in Section 5.5.4.

5.5.1 Benchmark Setup

Compared to the benchmark setup for the page based relocation (Section 4.3), the setup for the stack region write access leveling is slightly changed. The **fft** benchmark makes use of several vector processing functions. Making these functions compatible with the stack relocation requires a more complex implementation, which is out of the scope of this thesis. Just disabling the stack relocation during the vector processing functions may influence the result of the evaluation. Because of this, the **fft** benchmark is replaced by another benchmark, namely **pfor**. This benchmark decompresses a compressed set of numbers in small packets. For each packet the sum of numbers is calculated and stored in a global variable. This implementation simulates a realistic scenario, when a steam of compressed numbers arrives and has to be aggregated. For the compression, a lightweight compression, namely PFOR (Patched Frame of Reference) [31] is used. The benchmark reads the compressed data from a global array and decompresses a fixed amount of elements into a temporary local array. After the local array is aggregated, it is deleted (the local variable is removed in the next loop iteration) and the next set of elements is decompressed.

The **bitcount**, **lesolve** and **qsort** benchmark are also used to evaluate the stack region write access leveling. All benchmarks are modified to match the requirements for address consistency. Basically all benchmarks use the raw pointer adjustment and make only use of 64bit datatypes. Additional checks in the benchmark code make sure, that only 32bits of the variables are used. For all benchmarks, the correctness of the execution is tested by a detailed comparison of the application results with and without applied wear-leveling techniques.

5.5.2 Stack Only Leveling

Running an application with the stack region write access leveling basically enables three test cases. First of all, the application can be executed unmodified and all relocations are performed by the runtime system at a configured frequency during interrupt service routines. This execution is called **IRQ**. Secondly, the self adapting hinting mechanisms can be used to perform stack relocations at a good point of the execution. This run is called **Hinted**. Finally, the automatic mechanisms can be deactivated and relocations can be triggered only from the application itself. This run is called **Softonly**. In this evaluation, the last case is always implemented in a way, that the relocations are performed at the same point of the execution, where the hints were given before. Additionally, the amount of triggered relocations from the application is mostly the same amount of relocations, which are performed by the automatic mechanism[7]. The stack is relocated by an offset of 64 bytes on each relocation, because cache-lines (64 byte width) are assumed to be written entirely, thus a finer-grained relocation would cause no additional benefit.
Figure 5.6, Figure 5.7, Figure 5.8 and Figure 5.9 show the resulting memory write distribution for the 4 benchmarks in the three test cases, compared to the baseline (application execution without any wear-leveling). The gray lines in the plots indicate the boundaries of virtual memory pages (4kB). Please note the different scaling of the y axis. For each benchmark, several observations can be made:

- The **bitcount** benchmark (Figure 5.6) is an extremely simple application, which iterates over a data array in a loop and counts the bits set to 1. Thus, the application performs writes mostly only to the intermediate result variable and the loop variables. As the interrupt driven mechanism triggers a relocation of the stack after a fixed number of memory writes (the benchmark is configured to trigger a relocation on every 3000th write for the interrupt driven version and to request a relocation on every 4000th write for the hinted version to reach a similar total number of relocations), mostly the same number of writes is applied to each memory region of the

[7] For all benchmarks, hints are given at the beginning of loops or at the beginning of recursive functions. Achieving the same amount of relocations as the automatic mechanism does is done by performing the relocation with the same ratio, the adoptive mechanism figures out (after a certain threshold of loop iterations or at a certain recursion depth.)

stack accordingly. This leads t a mostly perfect line in the plot. The step, visible in the plot, results from the simple fact, that the stack movement does not end up at the left boundary of the stack memory when the application finishes. The results for the hinted and softonly executions have a similar character, but do not result in a totally even write distribution inside of the stack memory. For both experiments, additional code is inserted into the application, which is executed on every loop iteration. For the hinting mechanism, it can be seen that the execution of the adaptive hinting mechanism cause a high overhead, because the write count to the stack memory is significantly higher. For extreme small loop bodies, like in this benchmark, running the hinting mechanism in each loop iteration might be inconsiderable. In case the application directly triggers the relocations (softonly experiment), the overhead is reduced, because not the entire adaptive hinting mechanism has to be executed. However, the code for triggering the relocations changes the write behavior, because it also uses a local variable. This leads to a slightly worse result compared to the interrupt driven mechanism.

- In contrast to the **bitcount** benchmark, the **pfor** benchmark (Figure 5.7) is a more complex benchmark. For every iteration, a subset of compressed numbers is decompressed into a small local array and the decompressed numbers are aggregated. The benchmark decompresses 40 elements at once, thus the temporary array has 320 bytes (8 byte per element). Even with the additional variables, the application only uses a small part of the memory, which causes the small and high peak in the baseline experiment. Because of this, the interrupt driven relocation version (the interrupts are configured to occur on every 3000th memory write) again works out and in result mostly the same amount of writes is applied to all the stack memory. Because in this benchmark more work is done in a single loop iteration, the adaptive hinting mechanism does not cause such a high overhead as for the **bitcount** benchmark.

  It can be seen in the plot, that the benefit from the hinting (stack relocations are mostly performed when the volume to copy is small) mostly relativizes the caused overhead. Again, the additional executed code for the hinting changes the memory write distribution and results in a less even distribution after the stack relocations. For the softonly version, where relocations are directly triggered from the application, the result looks mostly the same like for the interrupt driven version. Again, due to the higher complexity of the loop body the inserted code to trigger the relocations has a smaller influence.

- The **lesolve** benchmark (Figure 5.8) differs from the previous benchmark in the point that the benchmark behavior strongly depends on the data. Depending on the structure of the equation system, the solver has to take more or less equations

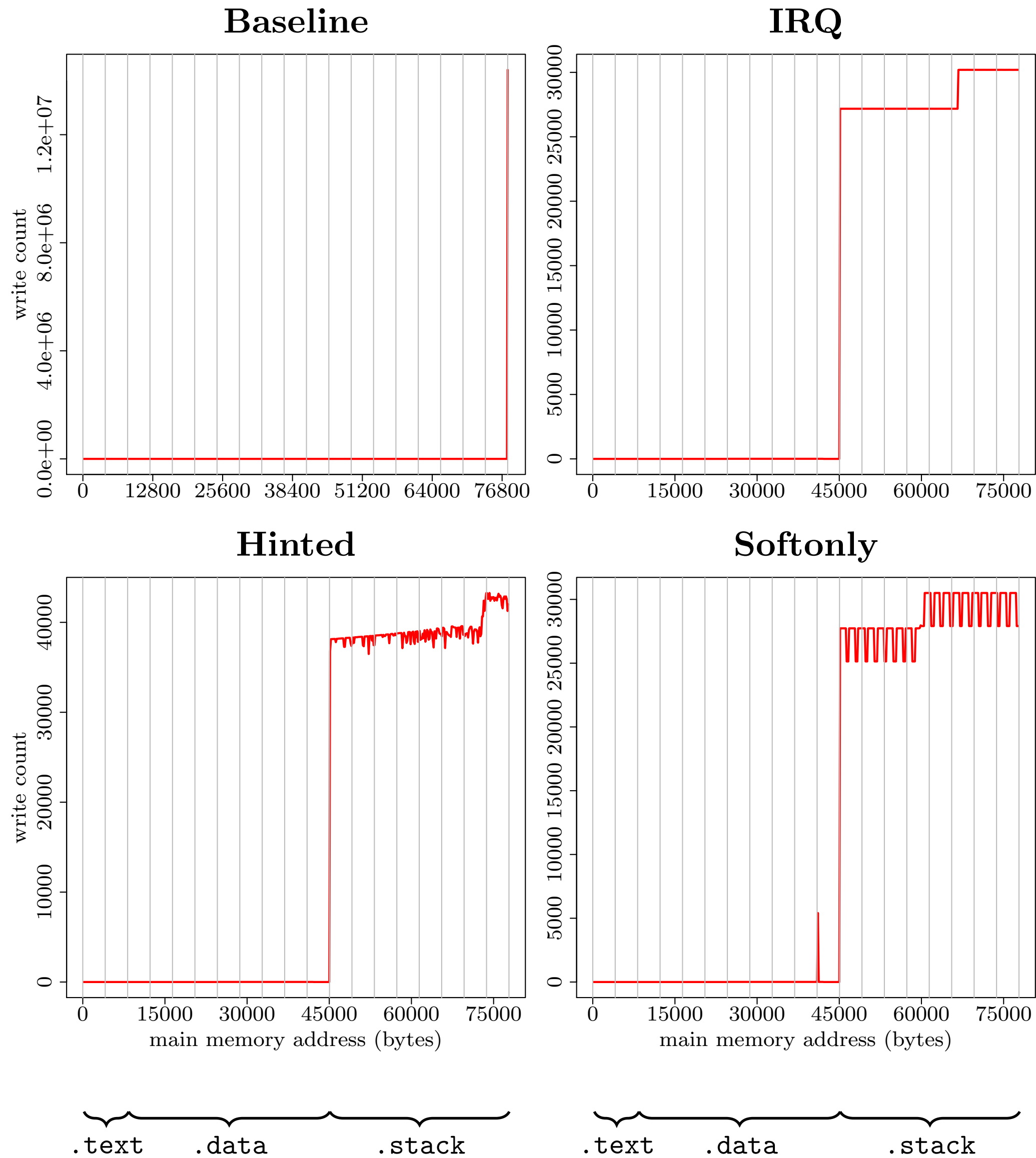

Figure 5.6: Stack Only Leveling - Bitcount Benchmark

into account to solve one equation. This does not have a big effect on the baseline, because the writes still only target a few bytes and cause a small and high peak.

Contrarily, for the interrupt driven version a certain non-uniformity in the write count to the different memory bytes of the stack can be observed. This results from the data dependent behavior, because even if the interrupt is triggered after a fixed number of writes (every 3000th write for this benchmark) the write distribution to the currently valid stack frame differs for each relocation. The results of the softonly run show, that the point of execution inside of the application, where the relocations are triggered, is chosen not optimal, because the resulting write count to the stack

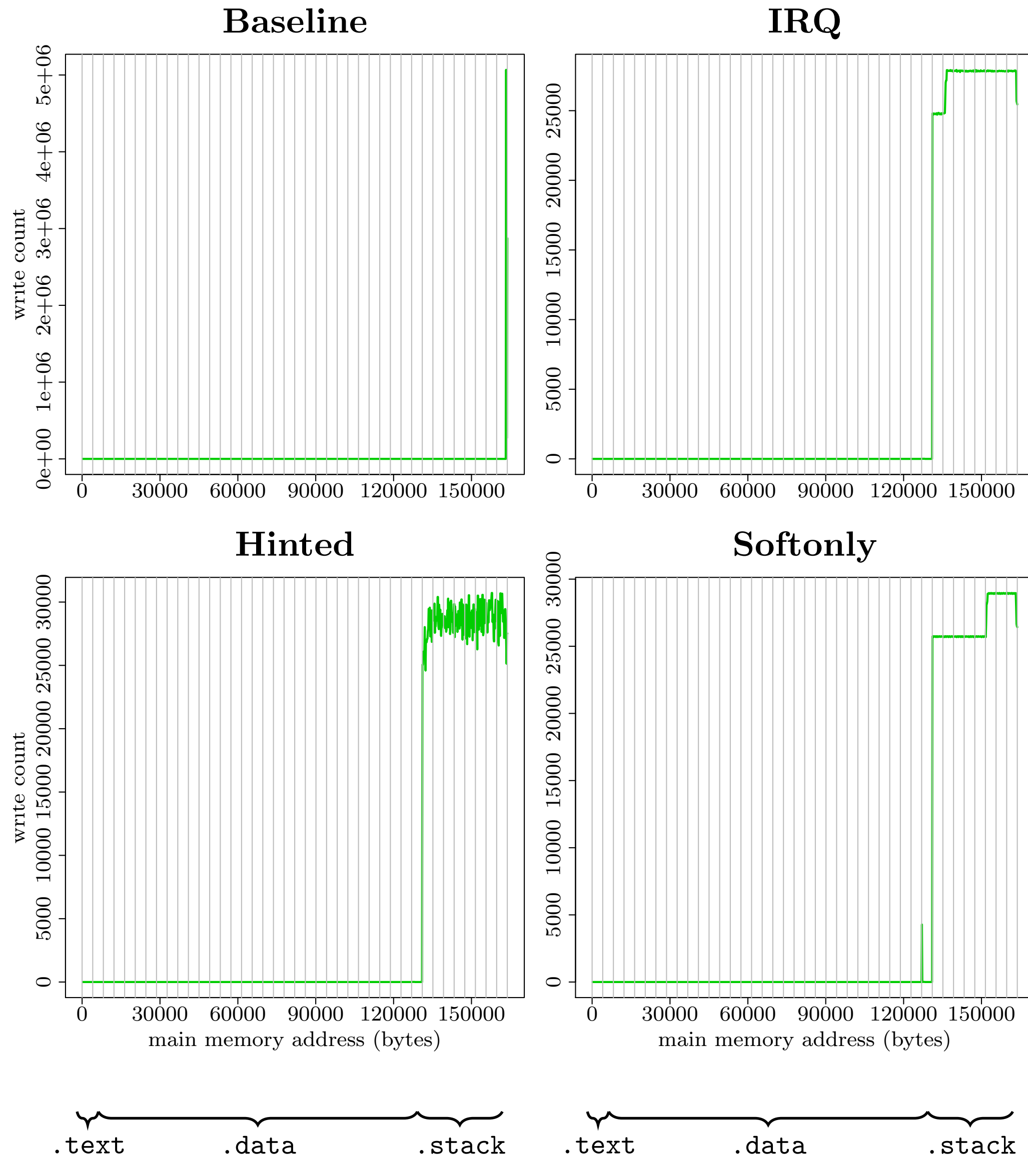

Figure 5.7: Stack Only Leveling - PFOR Benchmark

region is very uneven. As stated before, the manual triggered relocation is always performed at the same execution point, where the hint is given

The reason, the result from the hinted experiment is not similar to the softonly experiment, is that only 396 of 4728 relocations were successful hints. In consequence, the hinted version behaves mostly like the interrupt driven version. This indicates a bad placement of the hints.

- The **qsort** benchmark (Figure 5.9) is the only recursive benchmark and thus the only benchmark using the recursive adaptive hinting mechanism. For the baseline, the amount of used stack memory targets some memory pages, but in a non-uniform

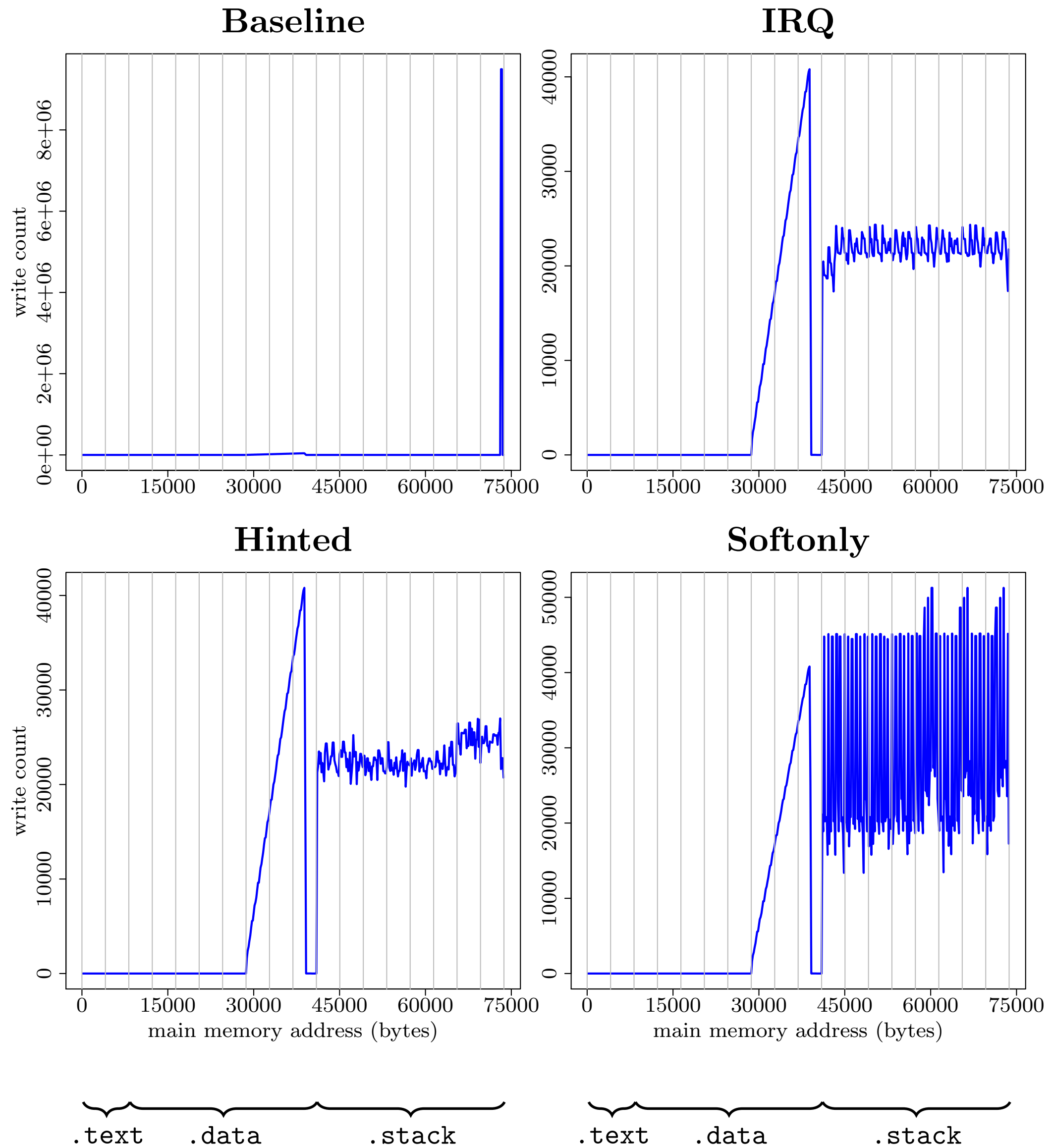

Figure 5.8: Stack Only Leveling - LESolve Benchmark

manner. Again, the write behavior to the stack depends on the data, because quicksort splits the array into two parts and performs recursive subcalls. Depending on the data, the size of the two parts is different and thus the recursion depth and number of calls is different. This results in a similar looking write distribution as for the `lesolve` benchmark. The adaptive hinting mechanism causes a big overhead in this benchmark, because most of the recurive calls only operate on a few data elements. The softonly version faces a big problem. The application spends some time with execution, which never causes a relocation. For the hinted version, relocations are triggered driven by the interrupt during this execution and the problem is resolved.

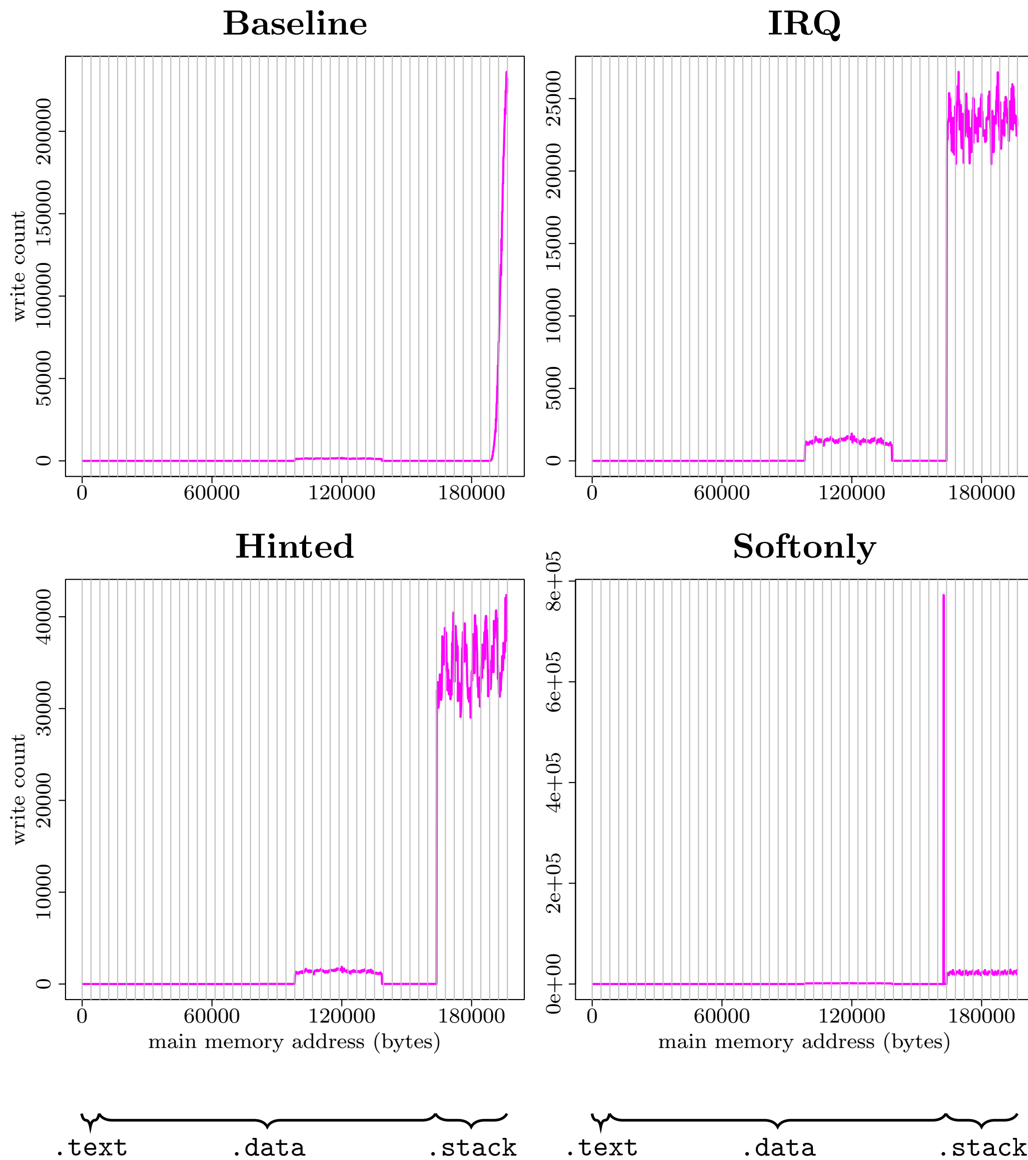

Figure 5.9: Stack Only Leveling - QSort Benchmark

Summing up, the evaluations of stack only wear-leveling show that software only wear-leveling for the stack region can be done and works out. Depending on the write behavior of the application the uniformity of write counts inside of the stack region varies but in all experiments the total available stack memory is used. Further more, the synchronous hinting environment turns out to be only considerable, when the work, performed by the application, between the given hints is big enough to relativate the overhead from the hinting mechanism. Performing stack relocations without runtime system support, like interrupt driven relocations and the hinting mechanism, turns out to be error prone, because the application easily might perform relocations at wrong points of the execution or perform no relocations at all, even if needed. As the stack region write access leveling

is not designed as a standalone technique, calculating metrics like achieved endurance (Equation (1.4)) on the benchmark results would not give useful insights.

5.5.3 Combined Leveling

Finally, the stack region memory write access leveling is an extension to the previously presented page based relocation. Thus, to evaluate the final achievement of wear-leveling, the stack leveling has to be evaluated in combination with the page based relocation. Section 5.4 gives the technical details of the integration of the stack leveling and the page based relocation. As already stated, the endurance tracking and the stack leveling share the same interrupt mechanism, which triggers an interrupt after a certain number of memory writes (3000 in the following evaluations). Thus, interrupt driven stack relocations are always executed together with a page relocation during the interrupt service routine. Every interrupt triggers a memory page to be relocated (threshold is set to 1) and a stack relocation, if not performed from the application before as a successful hint. Again, for each benchmark application three different versions of execution are considered. First, the stack relocation is only triggered by the interrupts, secondly the synchronous hinting environment is used and at last, the application performs the stack relocations manually. For all these executions, the page balancing is activated. The resulting write distribution is compared to activated page balancing without any stack leveling.
Figure 5.10, Figure 5.11, Figure 5.12 and Figure 5.13 show the resulting memory write distribution for the evaluation runs. Note again the different scaling of the y axis. The gray lines again indicate boundaries of virtual memory pages in the address space. For the different benchmark applications again several observations can be made and can be compared to the previous observations.

- The **bitcount** benchmark (Figure 5.10) already achieved a highly uniform write distribution to the stack memory for stack only leveling. For page based relocation and fully interrupt driven stack relocations, it can be observed that the uniformity inside the stack is maintained, but the regular relocations to other physical memory pages cause steps in the final distribution for every relocation. However, considering the total deviation, the different steps differ at most at 50 write accesses. The use of the hinting mechanism again turns out to cause a higher overhead (for details see Section 5.5.4) and achieve a more non-uniform write distribution. The softonly run causes some stack regions to be less written than others, which might result from the fact, that the relocations are not triggered frequently regarding the count of memory writes. This leads to a significantly worse allover result.
- For the **pfor** benchmark (Figure 5.11) slightly different observations can be made. The hinted and softonly runs turn out to achieve a better result than the fully interrupt driven run. This has two possible reasons. One is that the write pattern

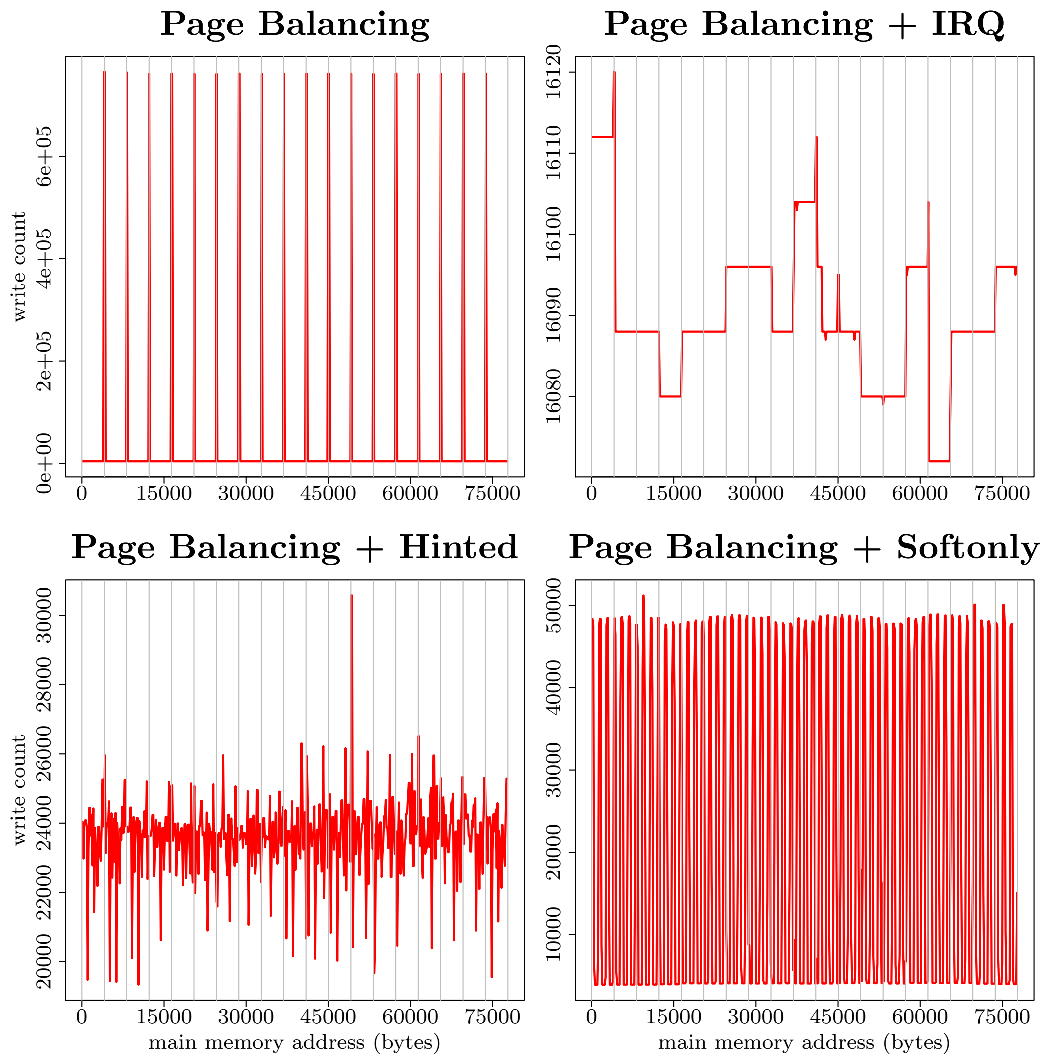

Figure 5.10: Page Relocation + Stack Leveling - Bitcount Benchmark

to the stack is more complex than for the **bitcount** benchmark, which has less influence on a circular movement, but if the physical location of the stack changes too, the resulting write distribution becomes non-uniform. Another reason is that the relative overhead of the hinting mechanism is not that big as for the **bitcount** benchmark, thus the hinting pays out more. For successful hints, the stack may be moved between page remappings, contrarily for the interrupt driven relocations the stack is always moved when the page mapping just changes. This results in the additional writes for relocations always to be performed to one page only and not to multiple pages. The softonly run shows that certain required relocations are missing, resulting in some high peaks.

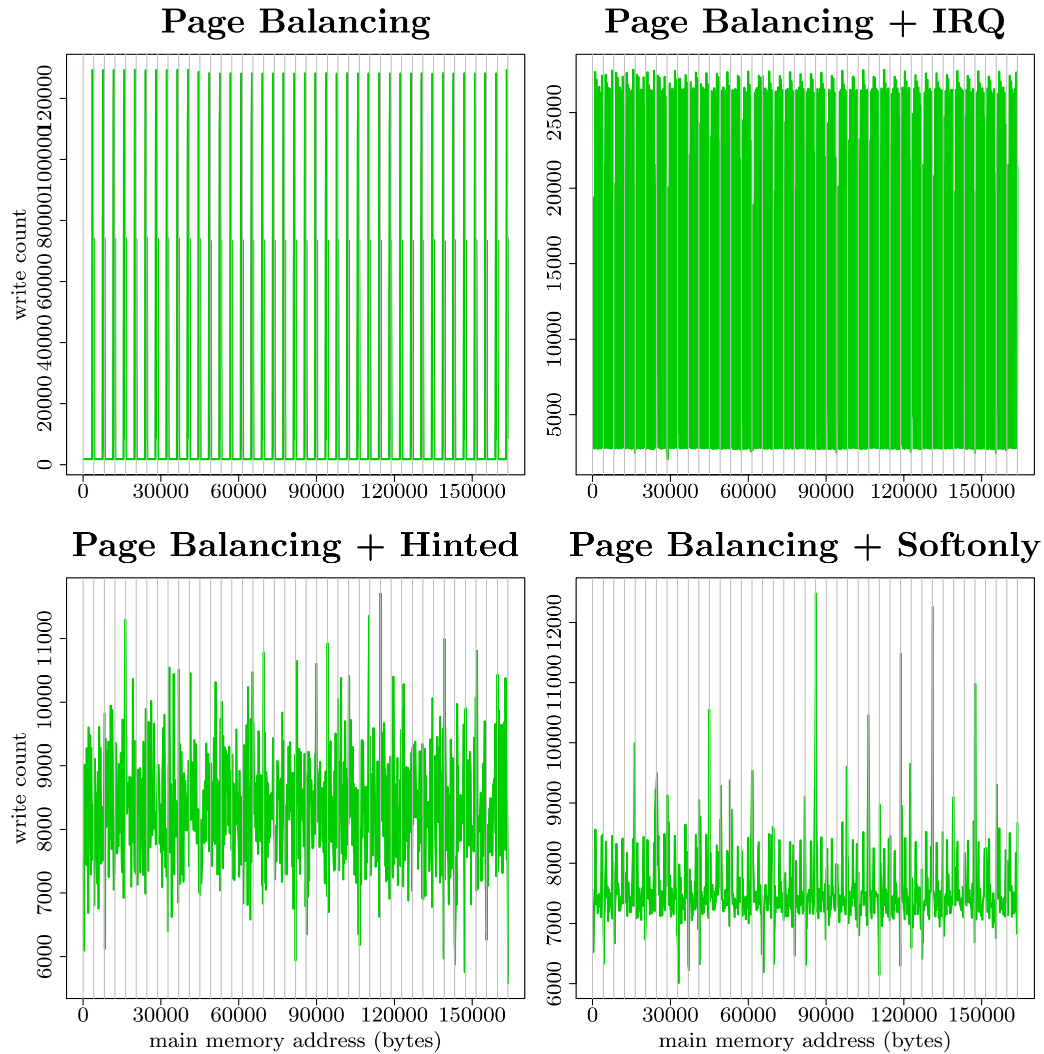

Figure 5.11: Page Relocation + Stack Leveling - PFOR Benchmark

- For the **lesolve** benchmark results (Figure 5.12) the results for all three configurations look very similar. The softonly run again shows some more high peaks, because relocations are not performed, when the interrupt driven mechanism would usually do.

- The **qsort** benchmark results (Figure 5.13) give a very interesting insight. While the fully interrupt driven version and the synchronous hinting environment are able to resolve at least some high peaks, the softonly version causes peaks, which are higher than without stack leveling due to a misplaced overhead. Already the results without page balancing (Figure 5.9) show that the softonly implementation spends some execution without without any stack relocation at all, causing the problem. Again for the interrupt driven and the hinted version the result looks very similar.

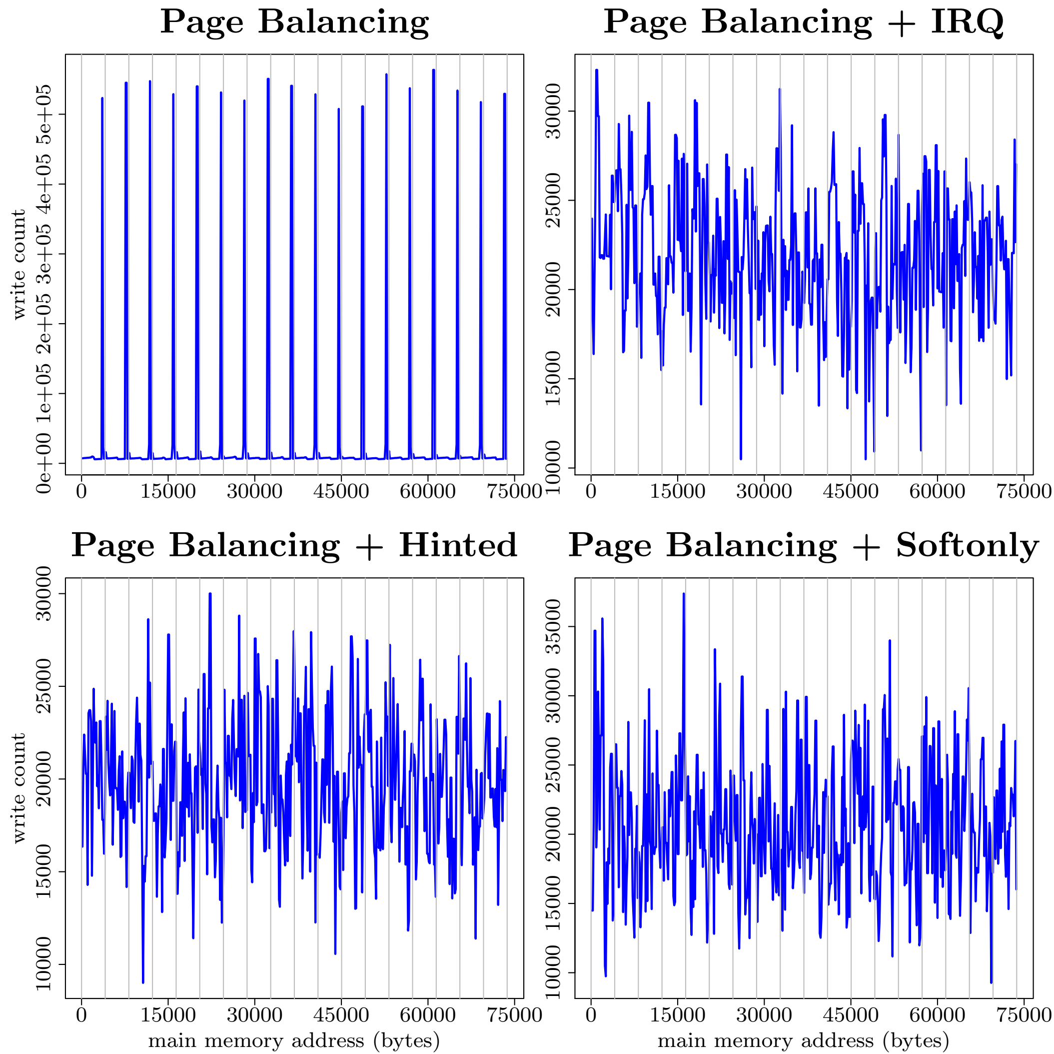

Figure 5.12: Page Relocation + Stack Leveling - LESolve Benchmark

To evaluate the resulting quality of the allover wear-leveling, the achieved endurance (Equation (1.4)) is calculated before, which is a metric of how uniform the total memory region is used. This metric can be calculated for the memory write distribution of an arbitrary benchmark execution. For the evaluation in this chapter, the achieved endurance is compared for following benchmark runs:

1. **baseline**: The execution of the benchmark application without any wear-leveling techniques. This version points out how efficiently the memory would be used regarding lifetime if typical applications are simply executed.

2. **page balancing**: Basically this is the final version from Chapter 4. The online endurance tracking system is used to identify hot and cold pages and swap them

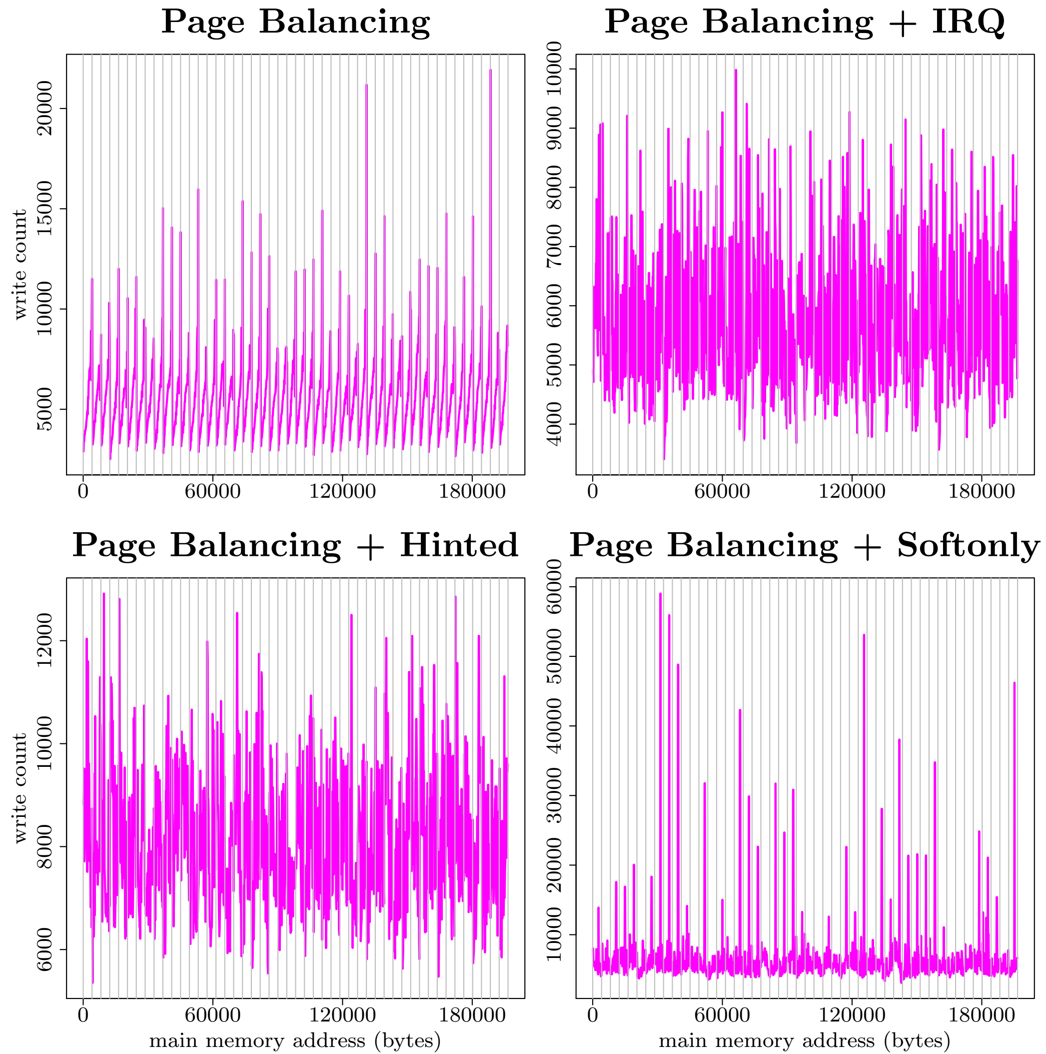

Figure 5.13: Page Relocation + Stack Leveling - QSort Benchmark

through the MMU accordingly. Any non-uniformity within memory pages is not resolved by this technique.

3. **IRQ**: In addition to the page balancing, in this run the stack balancing from this chapter is activated and fully triggered by interrupts triggered when exceeding a threshold of memory writes.

4. **hinted**: Instead of only triggering stack relocations by interrupts, in this benchmark run the synchronous hinting environment is used and hints are given a certain points of the application execution, where stack relocations are assumed to be cheap.

5. **softonly**: Finally the interrupt driven stack relocation mechanism is tuned off and the stack relocations are performed directly from the application.

Achieved Endurance of Combined Wear-leveling

For all these benchmark runs, the resulting memory write distribution can be found in the figures above. For all runs, except the baseline, the page balancing is activated. The endurance tracking is configured to record a sample every 3000th memory write, on every sample recorded the hottest page is relocated. The stack relocation system is also configured to relocate the stack on every 3000th memory write, for the **softonly** run relocations are triggered mostly as often as the interrupt driven version does.
Table 5.1 lists the resulting achieved endurance values for all benchmark runs. Note that the values strongly depend on the total amount of available memory. The more free memory is considered, the less the achieved endurance for the baseline is. In these evaluations, the amount of memory is limited to the required memory for the application to execute.

	baseline	page balancing	irq	hinted	softonly
bitcount	0.08%	2.08%	98.69%	71.13%	31.14%
pfor	0.1%	5.51%	26.12%	62.31%	57.67%
lesolve	0.13%	2.83%	50.39%	55.87%	43.8%
qsort	0.87%	21.27%	49.55%	56.96%	8.83%

Table 5.1: Achieved Endurance for Page Balancing and Stack Balancing

It can be seen from the table, that additional stack balancing, except the softonly version, improves the achieved endurance for all benchmarks significantly, compared to only page balancing. The observed high non-uniformity inside the stack regions is resolved up to a certain degree and enables the page balancing to achieve better results. Generally, the total wear-leveling setup offers a lot of configuration options, which can be tuned among different objectives. For this evaluation, page relocations were performed as often as stack relocations, which leads to a very fine-grained movement of write accesses through the memory. It could also be considerable to perform page relocations on a lower frequency than stack relocations to save some overhead of the page relocations. Contrarily it could also be considerable to perform stack relocations on a low frequency but with big relocation offsets to save more overhead. However, as already mentioned, exploring the full configuration space is out of the scope of this thesis.

5.5.4 Overhead

As already explained in detail in Section 4.4.3, considering the achieved endurance without considering the overhead does not give information about the improvement of the memory lifetime. The memory overhead for the total wear-leveling is caused by both wear-leveling systems. Page relocations cause the already explained overhead for copying memory pages

and maintaining the management information like the write approximation. The stack relocations additionally cause a memory overhead for copying the used stack content to a new location and for pointer adjustments. The synchronous hinting environments store some additional management information, which also causes a certain overhead. The memory write overhead is again determined using the total number of memory writes for a simulation run and compared to the baseline according to Equation (4.2).

	page balancing	irq	hinted	softonly
bitcount	34.06%	34.31%	83.6%	34.64%
pfor	34.07%	36.62%	37.16%	35.31%
lesolve	33.82%	36.4%	40.47%	37.2%
qsort	34.07%	42.18%	111.49%	49.79%

Table 5.2: Write Overhead for Page Balancing and Stack Balancing

Table 5.2 lists the calculated write overhead for the previously presented benchmarks. It can be observed, that the overhead is dominated by the page relocation and not by the stack relocation. This results from the fact, that the amount of copied stack is much smaller than a memory page for each stack relocation. The previous observation, that the hinting environment causes a big overhead for applications with small loop bodies can be also proved by the data. The **bitcount**, **lesolve** and **qsort** benchmarks show a higher overhead for the hinted version than for the irq or softonly version. For the **pfor** benchmark, the relation between processed executions inside the loop and given hints is different, thus the hints cause a relatively lower overhead.
Given the achieved endurance and the write overhead, the real improvement can be finally determined according to Equation (4.4). This number indicates, how much more executions from the application can be performed before the memory wears out, considering the overhead for wear-leveling.

	page balancing	irq	hinted	softonly
bitcount	19.39	918.49	484.27	289.1
pfor	41.1	191.19	454.29	426.21
lesolve	16.27	284.18	305.95	245.57
qsort	18.24	40.06	30.96	6.78

Table 5.3: Real Improvement (RI) for Page Balancing and Stack Balancing

The real improvement values are listed in Table 5.3. It can be observed, that for the **pfor** benchmark the hinting pays out, because it causes a small overhead and achieves a much better endurance. For the **lesolve** benchmark, the hinting also pays out, even if most of the hints are not successful. Summing up, the softonly stack leveling turns out to be error

prone and inefficient. It might save some overhead compared to the hinted version, but the result of the wear-leveling is generally worse (for the **qsort** benchmark even worse than without wear-leveling). Using the synchronous hinting environment pays out, if the hints do not cause a major overhead, i.e. the performed executions of the application between hints have to be enough. For the recursive **qsort** benchmark, the improvement of using stack relocation is not that high, because the stack usage is distributed over some memory pages, even without stack wear-leveling. Still, an improvement can be achieved.

5.6 Discussion

This chapter provides a detailed description of the stack region write access leveling approach and evaluates this technique as an extension to page based, aging-aware memory relocations in the end. Summing up, non-uniformity inside of the stack region is resolved by moving the stack periodically in a circular manner. Compared to the related work (e.g. the common allocator for stack and heap by Li et al. [22]) this approach is very different. Li et al. allocate a piece of memory for every subsequent function call and make the allocation process aging-aware. This approach faces two major disadvantages. First, the wear-leveling achievement depends on the application to make enough function calls and not to use memory too excessive in a single function call. Secondly, the earliest function calls may reside in the same memory locations for mostly all the execution time, which make the achievement of a uniform wear-level allover the memory hard. These two main arguments motivate the technique in this thesis to relocate the entire stack regularly, which is independent of application function calls and also reuses the memory regions of the earliest function calls.
The relocation of the entire stack requires some static operating system support to achieved the wraparound semantic and also can be executed totally without support from the application as an interrupt driven mechanism. The main problem is the consistency of pointers to stack variables, which become invalid on a stack movement. To keep pointers consistent, hard constraints are forced to the application. This thesis proposes two mechanisms to keep pointers consistent. Raw pointers, residing in the stack, can be adjusted during the relocation. Memory words are identified as a pointer by boxing the stack to a dedicated memory region. This permits the application to store any other memory word, which could be a pointer to the stack but has a different semantic (e.g. only put values using 32bit in 64bit variables). The other mechanism is to not use raw pointers at all. Instead a smart pointer is used, which adjusts to the stack relocation offset during dereferencing. Both mechanism require strong constraints on the application and generally require a reimplementation of the application. However, the evaluation shows that the stack relocation mechanism is able to extend the memory lifetime significantly which might make the required reimplementation of the application a considerable price.

Finally, this chapter introduces an additional hinting mechanism for the stack relocation mechanism, which aims to reduce the overhead for stack relocations. The concept is to perform stack relocations mostly when the currently valid stack is small and thus, the volume to copy is low. To achieve this, hints are added into the application source code and processed by the runtime system. If a hint is successful (a relocation can be performed according to the configured relocation frequency), the stack relocation is performed synchronously from the application at the point of the hint. During the evaluation, it turns out that giving the hints causes a certain overhead and should not be done too often. If the hints are given not too often and achieve a significant improvement (copy volume is saved), the hinting environment pays out and leads to better wear-leveling results. However, triggering relications only directly from the application without the frequency driven runtime system mechanism turns out to be bad in most test scenarios. Some parts of the execution miss stack relocations at all and the resulting write distribution is less uniform. All in all, the stack region write access leveling strongly depends on the application to be a considerable technique. If the application can be modified to meet the constraints for pointer consistency and has a highly non-uniform write distribution in the stack, the memory lifetime can be improved significantly. Giving additional hints (without executing the hinting code too frequent) also might help to improve the memory lifetime furthermore. The basic problem, that the lifetime of an application can shrink down to minutes on NVM, can be tackled by this approach and the lifetime can be increased again. The evaluations show, that for the benchmark applications the application lifetime could be improved by a factor of $\approx 300 - 500$ for most applications, even up to ≈ 900 for a special application, which uses an extreme small amount of stack memory.

6 Conclusion

The main purpose of this thesis is to propose software only wear-leveling techniques, which fill the gap when wear-leveling is required but no hardware support for certain wear-leveling techniques is available. All presented techniques, code excerpts and evaluation results are implemented and executed in a simulation environment, also introduced by this work. The techniques are applied to different benchmark scenarios and the correctness of the execution is checked. Stating all the detailed concepts, this thesis brings up some insights regarding software only wear-leveling techniques, which should be considered when software only wear-leveling is an option. To finally conclude this work, the presented techniques and insights are briefly summarized in Section 6.1 and Section 6.2. As the proposed mechanisms differ from the related work and state-of-the-art for in-memory wear-leveling, the related work is compared to the proposed technique afterwards in Section 6.3. Finally, software only wear-leveling is a wide field and allows further improvements or completely new approaches. A small subset of the visions, which could be realized on basis of this work, is given in Section 6.4.

6.1 Summary of Techniques

Due to the need of operating system support for the proposed wear-leveling techniques, this thesis includes a custom runtime system, which implements the necessary operating system support. Thus, the wear-leveling techniques can be implemented efficiently and evaluated in a simple test environment. As mentioned before, the main contribution of this thesis, apart from the simulation setup, are two software only wear-leveling techniques, one for coarse-grained wear-leveling and one for fine-grained stack wear-leveling.

The coarse-grained wear-leveling adopts the idea of using the MMU to modify the logical to physical address mapping with respect to the current aging level of the memory pages from literature. Usually the aging level is determined by a special hardware controller, which is not available in a software only solution. Thus, a statistical endurance tracking system is designed, which records an approximation of the memory write distribution during runtime on a configurable temporal and spatial granularity without the need for special hardware. The approximation then is used to support an aging-aware wear-leveling algorithm with aging levels of the memory. Technically, arbitrary, already existing aging-aware algorithms can be executed on top of the approximated data. For this thesis, an

RB-Tree based wear-leveling algorithm is developed, which selects the coldest page as a victim efficiently. Once the wear-leveling algorithm delivers a result, the virtual memory mapping is adjusted to exchange the physical location of the victim and the triggering page. As the content of the pages itself is also copied, the logical view from the virtual address space does not change at all, while writes target a different physical memory region.

The fine-grained approach extends the coarse-grained approach, because pages (4kB sized) are always relocated to 4kB aligned pages. A non-uniform write pattern inside of a page (e.g. only a few bytes are written) is relocated over the address space, but still results in a non-uniform allover write pattern. As this behavior is observed to mostly happen in the program stack, the fine-grained approach targets the stack memory. To do this in a software only manner, no aging-aware approach is applied, but the concept of start-gap wear-leveling by Qureshi et al. is adopted [26]. The entire stack is moved through a reserved memory region in a circular manner, which resolves the non-uniformity in a write pattern over time, if the write pattern does not change too fast. The required wraparound is achieved by a special virtual memory configuration, called shadow stack. The stack can be moved by a special assembly function, which can be called directly from the application or can be moved from an interrupt handler by the runtime system. As an overall parameter, the frequency and relocation offset is configured. A hinting environment allows the application to indicate good points to relocate the stack, which are processed by the runtime system according to the configured frequency. To maintain the consistency of pointers to the stack when the stack is moved, a smart pointer implementation, which adjusts on dereferencing, and a raw pointer adjustment is implemented. Both have to be considered and used appropriately in the application.

6.2 Summary of Insights

Applying and evaluating the previously described techniques points out some insights. First of all, both techniques do not change already existing structures (e.g. the memory allocator) inside of the operating system, thus they cause an additional overhead. One important insight is, that for some scenarios the overhead can be tuned arbitrarily. Of course, the configurable parameters (i.e. granularity of write distribution approximation, frequency of page and stack relocations, etc.) can be tuned among a certain overhead, which usually affects the quality of the wear-leveling directly. Contrarily, applications might tend to have a slowly or never changing write pattern. For instance in embedded systems, code is repeated regularly to build a control system. The memory write pattern of these systems, does not change much during a long time. In this scenario, not the frequency of wear-leveling actions is important, but the total number such that the entire memory space can be covered by relocations. Concluding from this, if an application

is known to run for a long time, the frequencies for page and stack relocations can be lowered and a lot of overhead can be saved. The resulting quality of wear-leveling does not change much compared to a more frequent relocation. Furthermore, the combination of coarse and fine-grained wear-leveling spawns a huge configuration space, which again allows configurations with low overhead and good results.
For the memory write distribution approximation, extra knowledge about the memory hardware can be integrated. For instance, if the endurance is not constant over all the memory, this can be respected in the data, the approximator passes to the wear-leveling algorithm. The wear-leveling algorithm then achieves a wear-level, respecting the endurance variation, without having knowledge about it. The page based coarse-grained wear-leveling generally turns out to suffer from non-uniform memory accesses within memory pages. Recursive algorithms cause more uniform accesses in the evaluation, because larger amounts of the stack are used. For simple programs, which are based on a single loop, only a few bytes of the stack are used and this achieves a bad wear-leveling result with coarse-grained page relocation only.
As a solution, the stack region writes access leveling helps in these situations and significantly improves the wear-leveling result. One important insight is that the constraints for pointer consistency require a lot attention while writing the application. Another insight is, that it is important to perform frequent stack relocations over time. If relocations are only performed directly by the application, important relocations are missing and the results turn bad. The hinting of relocations generally turns out as a nicely working system, but the usage inside the application has to be considered to not cause too much overhead.
Finally, important insights for the evaluation technique can be stated. The results of the evaluation depend on the amount of memory, which is considered for the evaluation. The memory usage of an application achieves worse analysis results, if unused memory is considered (e.g. a too large allocated stack region). In this thesis, the evaluations are performed with almost the minimal required memory for each application. When relocation based approaches are used, the memory overhead is an important effect to consider. Considering this for the memory lifetime is important, because at the end the important outcome is, how much more memory accesses, respectively executions the application can perform before the memory is worn out.

6.3 Comparison to Related Work

The important related work for this thesis can be divided by the two contributions, as already explained in detail in Chapter 2. The related works for coarse-grained wear-leveling usually use a hardware remapping mechanism to perform physical relocations [5, 17, 23] or use the MMU to perform relocations [14]. However, the aging of each page is usually gathered by additional hardware, which is the major difference to the technique

in this thesis. Up to my best knowledge, no comparable approach in literature exists. To overcome the need of additional hardware in literature, non aging-aware techniques are proposed. With the help of the proposed memory write approximation system in this thesis, also the aging-aware algorithms could be applied without additional hardware.
Related work for the stack relocation generally is rare. Some approaches for fine-grained wear-leveling exist [26, 27], but they do not target the stack memory explicitly. Instead they propose algorithms for generic fine-grained wear-leveling, which might be executed on additional hardware. Li et al. proposes to use a common memory allocator for heap and stack [21, 22], which allocates a new portion of memory for every function call. This technique can hardly be compared with the circular stack movement, proposed in this thesis.

6.4 Future Outlook

For future work, both proposed techniques can be optimized and extended with new features. Generally, the stack region write access leveling is an extension to the page based wear-leveling, only targeting the program stack. The other memory segments might still cause uneven memory accesses and should be leveled. Specialized solutions for the `data` and `bss` segment could also be considered.
For the page based relocation, some optimizations could be considered in future. The overhead of page relocations can be tuned by tracking currently used and unused pages. If a page was never used before, the content does not have to be copied on a relocation. Generally, the page based relocation is only analyzed for concrete tests applications in this work, the runime system should also be targeted by the page based relocation, which requires some extra implementation to synchronize the runtime system with th relocation. Additionally, the page relocation can be analyzed to work with multiple application threads on multiple cores.
For the stack region write access leveling also some improvements can be developed in future. Regarding the performance, the coarse and fine-grained wear-level can be synchronized, such that a page only is relocated, when the stack relocation totally passed the page, to not cause a higher memory usage in a subregion of the page. To improve the overhead, the hinting environment could also be improved. Applications could be analyzed with machine learning techniques to identify the best points to perform relocations. This would give a benefit for complex applications, where the hints cannot be placed manually due to the high complexity. Regarding the pointer consistency, compiler support to track and adjust pointers can be considered. Applications might still face constraints to allow the compiler to track pointers, but the constraints might be less invasive than the current ones. The compiler may provide a list of pointers to the runtime system, which can easily be adjusted, whenever the stack is moved.

List of Figures

1.1 Architecture of a DRAM Cell [20] 3
1.2 Architecture of a PCM Cell [13] 3
1.3 Architecture of a FeRAM Cell [24] 4
1.4 Programming a STT-MRAM Cell 5
1.5 Resetting a STT-MRAM Cell 5
1.6 Memory Write Distribution to Memory Regions 8

3.1 Architectural Overview of the Simulation Environment 20
3.2 Memory Placement of the Bootcode 21
3.3 Steps of the Initialization Code 21
3.4 Exception Level Overview 24
3.5 Class Overview of the Output Stream 25
3.6 Memory Placement of the Application by the Linker Configuration 30
3.7 Exception Level Changing while Interrupt Handling 31
3.8 Memory Write Trace of Four Benchmark Applications 33

4.1 Workflow of the Endurance Tracking System 38
4.2 Organization of the Virtual Memory Pages in a RBTree 43
4.3 Approximated Write Distribution - 1000 Writes Threshold (Left) and 5000 Writes Threshold (Right) 47
4.4 Slow Balancing (Endurance Tracking with Integrated Wear-leveling) 49
4.5 Fast Balancing (Endurance Tracking with Integrated Wear-leveling) 51
4.6 Simulated Write Distribution of Start Gap Wear-leveling on the fft Benchmark (Normal (Left) and 20 Times Extended Run (Right)) 55

5.1 Memory Layout of Relocated Stack 64
5.2 Shadow Stack 65
5.3 64bit Aligned Data Types 71
5.4 Smart Pointer Class 72
5.5 Call Graph for a Binary Recursive Algorithm 75
5.6 Stack Only Leveling - Bitcount Benchmark 83
5.7 Stack Only Leveling - PFOR Benchmark 84
5.8 Stack Only Leveling - LESolve Benchmark 85

5.9 Stack Only Leveling - QSort Benchmark 86
5.10 Page Relocation + Stack Leveling - Bitcount Benchmark 88
5.11 Page Relocation + Stack Leveling - PFOR Benchmark 89
5.12 Page Relocation + Stack Leveling - LESolve Benchmark 90
5.13 Page Relocation + Stack Leveling - QSort Benchmark 91

List of Source Codes

3.1 Example of Using the Output Stream 25
3.2 Systemcall Interface Function 26

4.1 Performance Counter Reset 37
4.2 Virtual Address Remapping 45

5.1 Synchronous Relocation - Stack Copying 67
5.2 Synchronous Relocation - Register Adjustment 67
5.3 Interrupt Relocation - Stack Register Adjustment 68
5.4 Interrupt Relocation - Stack Copying 68
5.5 Raw Pointer Adjustment in C++ 71
5.6 Smart Pointer Construction 73
5.7 Smart Pointer Dereferencing 73
5.8 Stack Memory Allocation 74
5.9 Smart Pointer Usage Example 74
5.10 Stack Relocation Hinting Interface 76
5.11 Stack Relocation Adaptive Loop Hinting 77
5.12 Stack Relocation Adaptive Recursive Hinting 78

Bibliography

[1] Arm architecture reference manual armv8, for armv8-a architecture profile. https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.

[2] Arm cortex-a53 mpcore processor technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0500e/BIIDADBH.html. Revision r0p3.

[3] Gcc online documentation. https://gcc.gnu.org/onlinedocs/.

[4] Gem5 wiki. http://gem5.org/Main_Page.

[5] H. Aghaei Khouzani, Y. Xue, C. Yang, and A. Pandurangi. Prolonging pcm lifetime through energy-efficient, segment-aware, and wear-resistant page allocation. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED '14, pages 327–330, New York, NY, USA, 2014. ACM.

[6] ARM. Primecell® uart (pl011). *Revision: r1p4 Technical Reference Manual*, 2005.

[7] ARM. Arm® generic interrupt controller. *Architecture version 2.0Architecture Specification*, 2008.

[8] ARM. Bare-metal boot code for armv8-a processors. *Application Note Version 1.0 Non-Confidential*, 2017.

[9] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song, and J. Xu. Hmtt: A platform independent full-system memory trace monitoring system. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, pages 229–240, New York, NY, USA, 2008. ACM.

[10] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, Dec 1972.

[11] S. Bhatti, R. Sbiaa, A. Hirohata, H. Ohno, S. Fukami, and S. Piramanayagam. Spintronics based random access memory: a review. *Materials Today*, 20(9):530 – 548, 2017.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[13] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao. Emerging nvm: A survey on architectural integration and research challenges. *ACM Trans. Des. Autom. Electron. Syst.*, 23(2):14:1–14:32, Nov. 2017.

[14] C.-H. Chen, P.-C. Hsiu, T.-W. Kuo, C.-L. Yang, and C.-Y. M. Wang. Age-based pcm wear leveling with nearly zero search cost. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 453–458, New York, NY, USA, 2012. ACM.

[15] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[16] J. Dong, L. Zhang, Y. Han, Y. Wang, and X. Li. Wear rate leveling: Lifetime enhancement of pram with endurance variation. In *Proceedings of the 48th Design Automation Conference*, pages 972–977. ACM, 2011.

[17] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé. Increasing pcm main memory lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 914–919, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[18] Y. Han, J. Dong, K. Weng, Y. Wang, and X. Li. Enhanced wear-rate leveling for pram lifetime improvement considering process variation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(1):92–102, Jan 2016.

[19] Intel. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3A:System Programming Guide, Part 1*, 2016.

[20] B. Jacob, S. W. Ng, and D. T. Wang. Chapter 7 - overview of drams. In B. Jacob, S. W. Ng, and D. T. Wang, editors, *Memory Systems*, pages 315 – 351. Morgan Kaufmann, San Francisco, 2008.

[21] Q. Li, Y. He, Y. Chen, C. J. Xue, N. Jiang, and C. Xu. A wear-leveling-aware dynamic stack for pcm memory in embedded systems. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2014.

[22] W. Li, Z. Shuai, C. J. Xue, M. Yuan, and Q. Li. A wear leveling aware memory allocator for both stack and heap management in pcm-based main memory systems. In *Proceedings of the 2019 Design, Automation & Test in Europe (DATE)*, 2019.

[23] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha. Curling-pcm: Application-specific wear leveling for phase change memory based embedded systems. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 279–284, Jan 2013.

[24] E. M. Philofsky. Fram-the ultimate memory. In *Proceedings of Nonvolatile Memory Technology Conference*, pages 99–104, June 1996.

[25] M. Poremba, T. Zhang, and Y. Xie. Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Computer Architecture Letters*, 14(2):140–143, July 2015.

[26] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 14–23, Dec 2009.

[27] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[28] Songping Yu, Nong Xiao, Mingzhu Deng, Yuxuan Xing, Fang Liu, Zhiping Cai, and Wei Chen. Walloc: An efficient wear-aware allocator for non-volatile main memory. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, Dec 2015.

[29] W. Zhang and T. Li. Characterizing and mitigating the impact of process variations on phase change based memory systems. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 2–13, Dec 2009.

[30] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM.

[31] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-scalar ram-cpu cache compression. In *Icde*, volume 6, page 59, 2006.

Eidesstattliche Versicherung
(Affidavit)

Hakert, Christian
Name, Vorname
(Last name, first name)

174833
Matrikelnr.
(Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present Bachelor's/Master's* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der Bachelor-/Masterarbeit*:
(Title of the Bachelor's/ Master's* thesis):

Memory Access Analysis and Endurance Leveling Approaches for Non-volatile Working Memory Systems

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Dortmund, 25.07.2019
Ort, Datum
(Place, date)

Unterschrift
(Signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Dortmund, 25.07.2019
Ort, Datum
(Place, date)

Unterschrift
(Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.