# Software-Based Memory Analysis Environments for In-Memory Wear-Leveling

*(Invited Paper)*

Christian Hakert, Kuan-Hsun Chen, Mikail Yayla, Georg von der Brüggen,
Sebastian Blömeke, and Jian-Jia Chen
Department of Computer Science, tu Dortmund, Germany
Email:{christian.hakert, kuan-hsun.chen, mikail.yayla, georg.von-der-brueggen,
sebastian.bloemeke, jian-jia.chen}@tu-dortmund.de

**Abstract— Emerging non-volatile memory (NVM) architectures are considered as a replacement for DRAM and storage in the near future, since NVMs provide low power consumption, fast access speed, and low unit cost. Due to the lower write-endurance of NVMs, several in-memory wear-leveling techniques have been studied over the last years. Since most approaches propose or rely on specialized hardware, the techniques are often evaluated based on assumptions and in-house simulations rather than on real systems. To address this issue, we develop a setup consisting of a gem5 instance and an NVMain2.0 instance, which simulates an entire system (CPU, peripherals, etc.) together with an NVM plugged into the system. Taking a recorded memory access pattern from a low-level simulation into consideration to design and optimize wear-leveling techniques as operating system services allows a cross-layer design of wear-leveling techniques. With the insights gathered by analyzing the recorded memory access patterns, we develop a software-only wear-leveling solution, which does not require special hardware at all. This algorithm is evaluated afterwards by the full system simulation.**

## I. Introduction

The ongoing research in the field of non-volatile memory (NVM) is distributed over a large amount of different topics and uses a variety of methods to evaluate and analyze the proposed solutions. Most of the research has in common that the targeted platforms do not exist. The reason is that the market rarely provides systems with NVMs and that the technical details about the available systems are difficult to acquire. Thus, researchers evaluate their approaches based on detailed models or simulations, which focus on the NVM property of interest. For instance, a model for detailed simulation of access latencies usually does not provide a detailed simulation of the physical aging effects inside of the memory cell.

The area of software based wear-leveling algorithms faces a similar issue. To evaluate a wear-leveling algorithm, the cell aging has to be determined for the execution of typical benchmark applications. Aging-aware wear-leveling algorithms also require the knowledge about the current cell aging during the execution as an input to the wear-leveling algorithm. To achieve this, the majority of works assumes the write count to a memory region to be provided by the memory hardware. This information is possibly processed by a physical model to determine a precise estimation of the cell age. Since hardware that provides a write count of memory regions is rarely available, these approaches are usually evaluated by a simulation of the memory write distribution after the application of the wear-leveling algorithm. The original memory access behavior of a benchmark application can be collected by a trace writing simulation for instance. Due to this evaluation methodology, several side-effects of the execution of the wear-leveling algorithm might not be simulated properly, e.g., cache pollution due to the algorithm execution, operating system interaction or timing related application behavior. To overcome this shortcoming, the simulation has to include as many components of the system as possible. For example, a simulation which also includes the memory subsystem with a cache hierarchy covers the cache pollution of the wear-leveling algorithm by default.

In this paper, we introduce a simulation setup to design and evaluate software-only wear-leveling algorithms. This implies that the algorithms do not require precise aging information from the memory hardware. Nevertheless, a functional simulation of such a hardware component could also be integrated into the simulation setup, which would allow the evaluation of several other wear-leveling algorithms in a similar way. The proposed simulation setup is based on the gem5 simulator [5], which is a cycle accurate full-system simulator. Hence, the hardware components of the simulated platform (e.g. MMU, interrupt controller, FP unit) are simulated on a functional level. This provides a sufficient amount of details to simulate the influences of an algorithm on the operating system and the underlying memory hardware. We combine the system simulator with the NVM simulator NVMain2.0 [15], which can be integrated into gem5, such that all memory accesses are forwarded to the NVMain2.0 implementation and can be applied to a physical memory model. Furthermore, NVMain2.0 allows us to collect detailed information about every memory access in a trace file, which can be analyzed subsequently.

Based on the memory trace file, we provide mechanisms to analyze the memory access behavior regarding the target of in-memory wear-leveling, respecting the changed memory access behavior due to the underlying operating system and hardware. Additionally, we isolate the analyzed application from the operating system, which enables a separate analysis of the application and the operating system. The novel contributions of this paper are:

- The simulation setup (gem5 + NVMain2.0), which simulates the target hardware platform on a functional level.
- Analyzing methodology for the output trace files of the simulation setup, which analyze the memory access be-

978-1-7281-4123-7/20/$31.00 © 2020 IEEE

havior regarding in-memory wear-leveling, i.e. the write distribution of the application over the memory space.

- A case study on the design of a software-only in-memory wear-leveling algorithm, including the extraction of design targets from an initial application memory behavior analysis and an end-to-end evaluation of the algorithm, deployed in the simulation setup.

This paper first provides a brief overview of the related work, describing existing NVM simulators as well as the methodology used to evaluate in-memory wear-leveling algorithms in Section II. Subsequently, the technical details about the simulation setup are discussed in Section III. After demonstrating the usage of the simulation setup in the case study of an in-memory wear-leveling algorithm in Section IV, the paper gives an outlook on performance improvements (Section V) and concludes in Section VI.

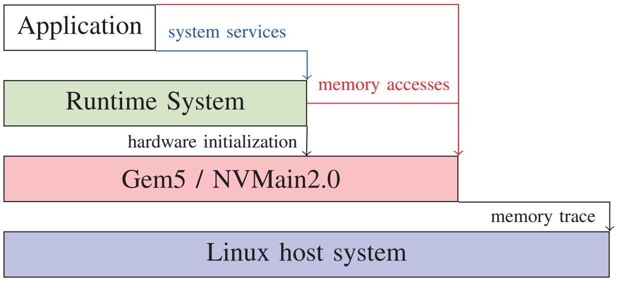

Fig. 1. Overview of the simulation setup

## II. Related Work

Several publications target the area of NVM system simulations due to the unavailability of real systems. Besides low level simulation on the cell or circuit level [8], [11], [14], [18], NVMs are simulated from the CPU perspective of read and write accesses in [3], [11], [15], [17]. These simulations usually require to capture the memory accesses from the CPU and pass it to the NVM simulator. This can be achieved in hardware, either by plugging special hardware into the DIMM slot of a PC [3] or by using advanced hardware architectures which combine a CPU and FPGA in one chip [11]. Alternatively, the NVM simulation can be implemented entirely in software and process memory access information from system simulators [15]. Regardless of the chosen simulator implementation, these simulations usually focus on a precise simulation of the NVM regarding timing, energy consumption, etc. The memory access behavior of the application and the resulting effects are only considered indirectly.

For the research in the area of in-memory wear-leveling, one of the most important aspects is the memory access behavior of the application, respectively the resulting memory access behavior of the final setup including wear-leveling. Therefore, recent publications do not focus on precise NVM simulations, but acquire the application's memory access behavior and apply it to in-house simulations of the wear-leveling algorithm [10], [12], [13], [16], [19]. This method usually lacks a precise simulation of the executing environment, including interactions of the application with the system and effects of the wear-leveling on the application execution.

To bridge this gap, in this paper we proposes a simulation setup that analyzes the memory behavior of an application, including applied maintenance algorithms like wear-leveling, but also simulates the entire executing system in detail, including a simulation of the NVM.

## III. Simulation Setup Architecture

The main purpose of our simulation setup is to allow a precise analysis of the memory access behaviour of an arbitrary application. Hence, it must include all interactions of the application with the operating system services and the underlying hardware, as they also influence the memory access behaviour. To achieve this, our simulation setup centrally consists of a cycle-accurate full-system simulator, i.e. gem5 [5], which provides the functional simulation of the underlying hardware. As the main application of this simulator, we execute a lightweight runtime system, which is equipped with the essential operating system services the application requires to be executed. The subsequent execution of the target application inside of the runtime system then results in a precise simulation of the memory behaviour of the application in a real-world system. Figure 1 illustrates the orchestration of the three main components (gem5, runtime system, and application) to the final simulation setup. It is shown that memory accesses are caused directly by the application as well as indirectly by the runtime system. Both types of memory accesses are processed by the simulator and captured in a memory trace, which represents the memory access behaviour of the application.

Besides capturing the memory accesses, an application usually performs, a meaningful analysis of these accesses is highly desired. For this purpose, the application's memory can be separated into the distinct memory segments the application uses, i.e., `stack`, `text`, `data`, and `bss`. Our simulation setup applies several isolation techniques to ensure that accesses of the application to each of these specific segments target a separate part of memory. As a result, the memory accesses can be identified by their access address later on. Furthermore, this allows a separate analysis for each memory segment. In addition to the applied isolation, the traced memory accesses can thus be processed and evaluated regarding different targets. For the purpose of wear-leveling for instance, the accumulated write counts to fixed-sized memory blocks needs to be determined. For the purpose of resource allocation planning, the memory consumption over time might be a very important aspect to consider. Our simulation setup processes the captured memory trace according to the desired target. The result of this process is a summarized representation of the important memory access behaviour of the target application.

The scope of this section is to point out the technical details of our simulation setup. The technical architecture is presented in Section III-A. Subsequently, the implemented isolation techniques and the impact on the simulation are described in Section III-B. This section concludes with an overview of the processing of the recorded memory traces in Section III-C.

### A. Technical Architecture

As shown in Figure 1, the simulation environment consists of four layers:

1) The host system (a powerful linux server).
2) A gem5 / NVMain 2.0 instance.
3) The lightweight runtime system.
4) An arbitrary application.

For completeness, we provide technical details of the four layers:

1) The choice of a Linux machine as the host system is due to the compatibility with gem5 and the possibility to comfortably process the results from gem5.

2) The cycle-accurate simulator represents the second layer of the simulation setup together with the memory simulator. In this setup, we use the full system mode of gem5 to be able to also capture system calls and operating system services in the resulting memory trace precisely. Due to the support in gem5, we choose the ARM 64 bit implementation as the simulated CPU. The CPU model is the DerivO3 CPU, which is the most advanced CPU offered by gem5 [5], including pipelining and out-of-order execution. As the simulated machine, we apply the VExpress_GEM5_V2 machine, which offers commonly available components in ARM-based embedded systems, like an interrupt controller and a uart controller. The NVMain 2.0 simulator is deployed as a plugin for gem5 [15]. All memory accesses issued during the simulation are forwarded to the NVMain2.0 implementation. NVMain2.0 applies all accesses to a model of the underlying NVM hardware, which includes, for instance, latencies and energy consumption. Furthermore, NVMain2.0 allows to write a trace of all memory accesses. The trace file includes the memory controller cycle, the access type (read or write), the access address, the old memory content and the new memory content for each memory access. This configuration allows us to capture all memory accesses and to collect precise information about each of these accesses for a subsequent analysis.

3) As the third layer of our simulation setup, we deploy a custom runtime system as the main control flow, executed in gem5. Due to the full system mode of gem5, the simulated program has to perform several operating system services, like hardware initialization, interrupt handling and memory allocation. As stated before, the memory accesses caused by all these services are also part of the simulation result, since they are executed inside of the simulated program. Furthermore, as the runtime system only implements the necessary services in a simple way, it is easy to extend and evaluate new operating system services, for instance to perform wear-leveling for NVMs. As a last important aspect, the custom runtime system implementation allows a precise control of the memory placement. Not only special, separated addresses can be chosen for memory allocations during runtime, but also different memory segments can be placed to separated memory regions during the linking of the runtime system. As these memory segments can be identified by their access address in the simulation result (i.e., the memory access trace), the memory behaviour of different operating system services or the different memory segments of the application can be analyzed separately. To achieve this, the runtime system introduces additional symbols into the compiled binary, which can be identified by their memory address in the binary. These symbols are used as markers for the beginning and ending of memory regions of interest[1].

4) The fourth layer of the simulation setup is the analyzed application. The application is compiled into the same binary as the runtime system, which saves the overhead for loading the application. After the runtime system initialized the hardware and set up the required drivers and services, the entry point of the application is executed. The runtime system applies further isolation techniques to separate the memory accesses from the runtime system and the application to allow a distinct analysis of them. The application can request operating system services by performing the according system calls or by directly calling library functions, provided by the runtime system.

### B. Application Isolation

To clearly distinguish between memory accesses from the runtime system and memory accesses from the application, both have to target distinct, identifiable memory regions for every access. To achieve this, the runtime system applies a generic spatial isolation. Additionally, an interrupt isolation is applied to separate the stack from the runtime system and from the application.

*1) Spatial Isolation:* As explained before, the runtime system has the ability to control the memory layout of the loaded binary during the linking process. This allows the runtime system to place the `text`, `data`, `bss`, and `stack` segments of the application into different memory regions, separated from the according memory segments of the runtime system. This method ensures that the application can be analyzed separately from the runtime system. In consequence, this allows an isolated analysis of the memory accesses caused by the application code. Therefore, the memory behaviour of the application can be analyzed separately from the operating system.

*2) Interrupt Isolation:* The aforementioned spatial isolation applies a static separation of the memory regions, but does not ensure that all memory accesses target their specific region during runtime. For compiler generated code this can be assumed due to the specification, but for the interrupt implementation of the runtime system an additional technique has to be applied. Interrupts may be driven by external timers or other sources, which request an interrupt during the execution of the application. The usual procedure to handle the interrupt is to save necessary CPU registers on the stack, to be able to resume the execution of the interrupted application. For this purpose, the current stack pointer is normally used, which implies that the interrupt handling causes memory accesses to the application's stack, even if the interrupt handler belongs to the runtime system. To avoid this problem, we use an ARMv8 specific feature that allows to use different exception levels, where a separate stack pointer can be assigned to each exception level [1]. The assignment of the lowest exception

[1] To unambiguously identify a memory region by the traced access address, the remapping of the memory (virtual memory) has to be respected. Either the mapping is known at every time (by logging changes) or the runtime systems applies an identity mapping only.

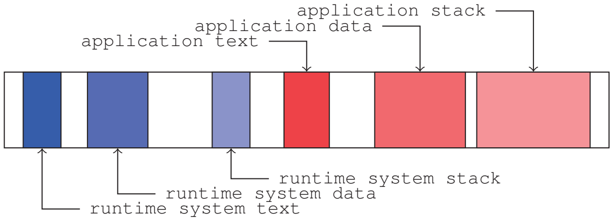

Fig. 2. Exemplary memory layout of a simulation instance

level (EL0) to the application and a higher exception level (EL1) to the runtime system causes every interrupt during the application to be handled on EL1. The hardware automatically switches to the stack pointer of EL1, thus the runtime system's stack is used to save the registers[2]. Applying this method, all memory accesses due to interrupt handling are guaranteed to target the runtime system's stack and not the application's stack.

The combination of both isolation techniques makes sure that the memory accesses of the application and the runtime system are completely separated and thus can be analyzed separately.

### C. Trace Processing

After the isolation techniques are applied, the resulting memory trace has to be analyzed. NVMain2.0 provides the following information for every memory access in a trace file: < Memory controller clock cycle, Read or write access, Access address, Old memory content, New memory content, Causing CPU core >
Due to the aforementioned separation of the memory, the access address can be used to identify the causing software element of the memory access. In a first step, this is used as a filter to discard memory accesses, that are not meaningful for the analysis. For instance, when the purpose of this simulation setup is to analyze the application behaviour, the memory accesses of the runtime system are not important and can be discarded. Furthermore, only the memory behaviour of a benchmark phase might be desired to be analyzed and a start-up and initialization phase can be discarded. Figure 2 illustrates an exemplary memory layout of the simulation setup.

After filtering out unimportant memory accesses, the remaining accesses should be aggregated into a meaningful summary of the memory usage behavior. As this paper targets the purpose of in-memory wear-leveling, we focus on aggregating the write-count to specific memory regions, since the aging of memories is influenced mainly by the total number of writes[3]. For the aggregation of the write accesses, the memory architecture has to be taken into account. For each write operation, an entire cache-line is written to the memory. In our case, the cache-line size is 64 bytes. Hence, during the analysis, we increase the write-count of the entire 64 byte cache-line for every write. After the aggregation, the write-count distribution can be illustrated graphically. The graphics can include the information about the different memory segments as well, which allows an interpretation of the memory usage behaviour of the distinct memory segments. Figure 3 illustrates a graphical representation of the analysis result. The memory space (x axis) is indicated with the distinct memory segments, thus the write-counts can be distinguished. Furthermore, the illustration indicates the boundaries of 4 kB virtual memory pages with grey vertical lines. The results originate from the execution of a simple bitcount benchmark and a data decompression benchmark, using the lightweight patched frame of reference compression [20]. In this example, an intensive memory usage at the top of the stack can be observed.

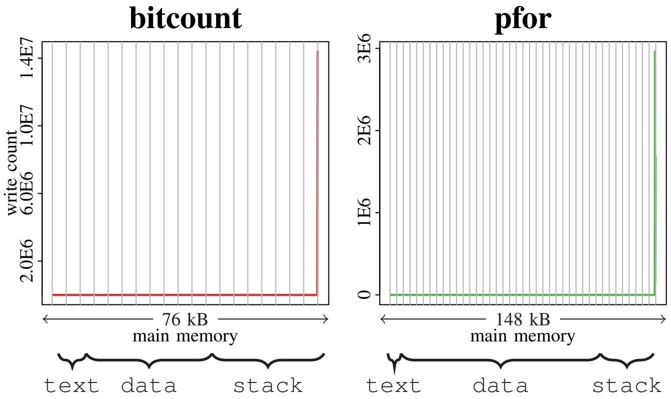

Fig. 3. Exemplary memory write-count aggregation analysis

## IV. Case Study: Program Region Analysis and Deduced Wear-Leveling Schemes

Figure 3 gives an example of how the proposed simulation setup can be used to analyze the memory access behaviour of an application in the context of in-memory wear-leveling. This section intends to make further use of the simulation setup features to provide a detailed analysis of the application's memory write access behaviour. Based on the observed behaviour, a wear-leveling scheme is deduced, implemented and again evaluated with the simulation setup. Many of the wear-leveling mechanisms proposed in the literature are evaluated based on specific models or simple simulators, but an end to end evaluation, where the wear-leveling algorithm is deployed to a real or at least fully simulated system, is rarely provided. Our simulation setup offers the possibility to analyze the access behaviour of an applied wear-leveling algorithm in the same way as the access behaviour of the bare application. Thus, the impact of the wear-leveling mechanism can be analyzed in detail and with respect to all influences of the execution of the mechanism on the target system.

In this section, first a set of benchmark applications is executed in our simulation environment and analyzed in the context of in-memory wear-leveling in Section IV-A. After this, a wear-leveling algorithm is proposed to tackle the observed situation and presented in Section IV-B. Subsequently, the end

[2] A similar technique could also be applied without special hardware support when the interrupt handler switches the stack pointer first. However this would require to store the other stack pointer in memory.

[3] For some NVM technologies, it has been shown that not only the number of writes is important for the cell aging [7], but also for instance the time interval between writes. However, the further required information for the advanced cell age determination can usually also be extracted from the memory trace, by applying according physical models.

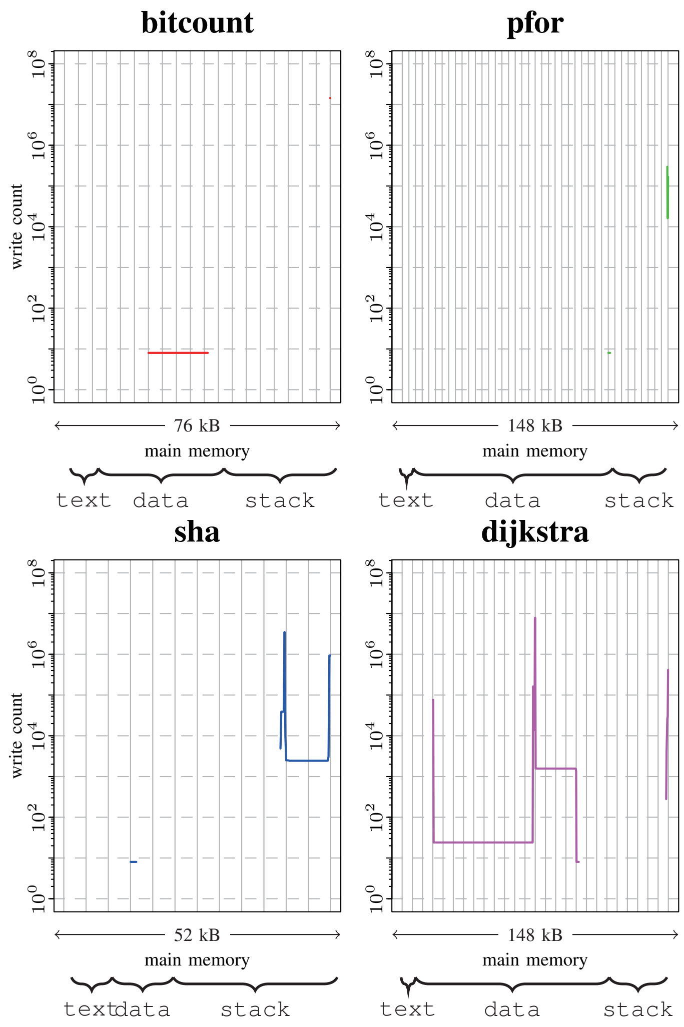

Fig. 4. Memory write behaviour of four benchmark applications

to end analysis is performed and the resulting memory access behaviour of the simulation of the wear-leveling algorithm is provided in Section IV-B.

### *A. Benchmark Application Analysis*

To get an overview of the memory access behaviour regarding writes of some embedded applications, we executed four benchmark applications in our simulation environment. The simple bitcount and data decompression benchmark are already presented in Figure 3. The bitcount application iterates over an array of numbers and counts the 1 bits of these numbers. This is done by a static set of instructions. The data decompression benchmark (pfor) processes batches of compressed numbers, decompresses them locally and aggregates them afterwards. The other two benchmarks (sha and dijkstra) are taken from the mibench security and networking suite [9]. All benchmarks are executed in the simulation environment and the write accesses of the application are isolated in the postprocessing. Figure 4 illustrates the resulting write-count distributions graphically. Note that due to the logarithmic scale of the y axes a value of 0 is not shown in the plots.

The results of the memory analysis can be collected for every memory segment and interpreted separately:

- `text`: As the text segment contains the compiled binary CPU instructions, it is only read during the program execution. For wear-leveling analysis, the text segment is less important, because it is never written. The wear-leveling algorithm only has to make sure that the physical memory from the text segment contains a logical segment, which is written from time to time.
- `data / bss`: The data and bss segments both store global variables and objects from the application. The usage of these variables highly depends on the application. For the pfor benchmark, we observed that the data and bss segments are not used for writing data at all. The bitcount, sha, and dijkstra benchmarks only write to a small part of the data and bss segment. Nevertheless, this part is written intensively and has to be taken into account for the wear-leveling decision.
- `stack`: The stack segment is the most interesting segments in the context of wear-leveling. It is used for local variables and objects, as well as to store necessary information for each function call. As a result, the write behaviour to the stack is not completely controlled by the application, but also by the compiler generated code for function calls. Our results show that the write pattern to the stack is different for each of the analyzed benchmark applications. Generally, the applications write intensively to small parts of the stack, while other parts of the stack are never written. The part of the heavily written stack memory also differs for the applications.

In summary, the analysis results of the four benchmark applications point out two key needs for a wear-leveling mechanism. First, the wear-leveling mechanism has to be aware of the not written memory regions and has to redirect write accesses to these regions to ensure they are used equally often as the written memory regions. Second, the wear-leveling mechanism has to be aware of the small but intensively used memory regions, especially inside of the stack segment. The wear-leveling algorithm has to recognize the write-count to these memory regions, or a similar metric, to swap the memory regions in a way that the total write-count to each memory region incrementally reaches the same level. This requirement makes the wear-leveling algorithm aging-aware. The incremental leveling of the write counts also omits the need to persist a state of the wear-leveling algorithm during reboots.

### *B. Page Swapping Wear-Leveling*

As a demonstration in this paper, we apply a fairly simple wear-leveling algorithm to the previously mentioned scenario. The algorithm is purely software-based, which means no additional hardware is required to perform the wear-leveling. Therefore, we do not have to equip the simulation environment with a simulation of the additional hardware. The details of the algorithm implementation are briefly stated subsequently.

First, the wear-leveling algorithm achieves the previously mentioned aging-awareness with a runtime sorting approach. During runtime, the current memory age is estimated by sampling write accesses to the memory. The sampled write accesses are aggregated to larger memory regions and managed in a data structure. We utilize a sorted RB-Tree [4], hence

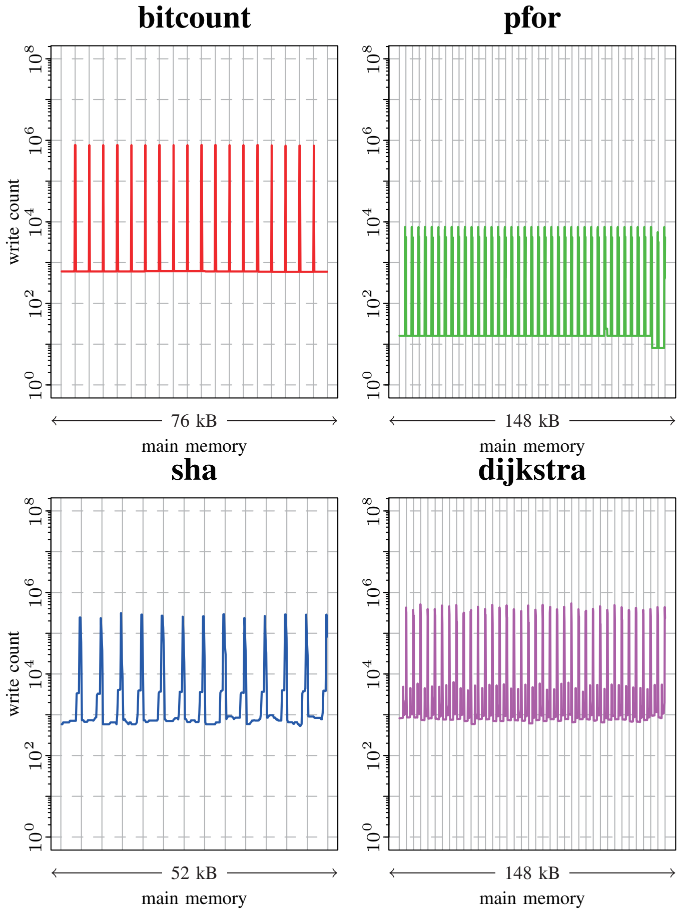

Fig. 5. Simulation-based evaluation of the wear-leveling algorithm

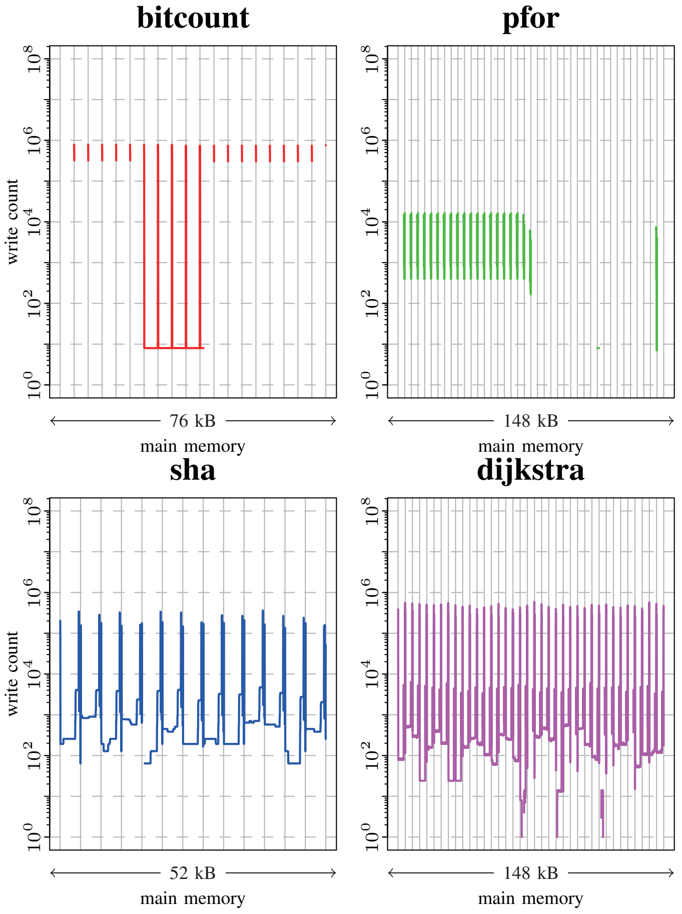

Fig. 6. High-level evaluation of the wear-leveling algorithm

the currently least written memory region can be determined easily. The algorithm periodically decides to exchange a memory region which was heavily written in the last period with the least written memory region. Therefore, the currently least written memory region likely is targeted by writes in the next period and will not be the least written region. Hence, the too heavily written logical regions target all physical memory regions over time. This achieves the incremental leveling of the number of writes over all memory regions.

Second, whenever the algorithm determines a logical memory region to be swapped to another physical memory region, the actual swapping is performed using the virtual memory pagetables. By swapping the physical mapped page of two virtual pages and copying the content from one physical page to the other, the logical perspective of the application is maintained. After this operation, the writes to one virtual memory page target the other physical memory page and vice versa, which precisely achieves the required wear-leveling action. The usage of the virtual memory subsystem limits the algorithm to a granularity of virtual memory pages, i.e., 4 kB, which makes the approach coarse-grained. Nevertheless, this approach is a generic, aging-aware, software-only wear-leveling algorithm.

### C. End-to-End Analysis

To demonstrate the purpose of analyzing an algorithm with the proposed simulation setup, we execute the previously described wear-leveling algorithm on the benchmark applications shown in Figure 4. The memory write behaviour is captured with the same mechanism as before, but the application in contrast contains the wear-leveling implementation in addition. Hence, the resulting plots in Figure 5 contain all interactions of the wear-leveling algorithm with the runtime system and the hardware, as for instance the MMU driver, the pagetables, and an interrupt mechanism to trigger wear leveling actions[4]. Figure 5 does not display the wear-leveling result under several assumptions about the execution of the wear-leveling algorithm, but displays the real memory trace of the executed wear-leveling algorithm in contrast. In comparison, Figure 6 shows a high-level evaluation of the implemented wear-leveling algorithm. The application's bare memory trace (Figure 4) is processed by a high-level model of the wear-leveling algorithm, which tracks the internal data structures and maintains an artificial virtual memory remapping table.

Focusing on the wear-leveling algorithm itself, the analysis results show that the wear-leveling works under the given conditions. Due to the granularity of virtual memory pages, the result contains non-uniform patterns within the single pages, but the incremental leveling of the write-counts is achieved.

Focusing on the difference between the high-level and simulation-based evaluation, the comparison shows various

[4] Note that the mentioned interactions only have partial influence on the visible plots, since the plots are already filtered and shrinked to the application's memory size.

differences. First, the timepoints of the wear-leveling actions are slightly shifted, as the high-level evaluation does not take overheads, e.g. for interrupt handling, into account. Thus, the intra-page write patterns differ between the two results. Second, the model for the sampling mechanism of the write accesses lacks of accuracy, leading to a slightly different number of wear-leveling actions. This can be observed in the evaluation of the pfor application.

To summarize, the simulation setup presented in this paper is used to analyze four benchmark applications and process their memory write behaviour in the context of in-memory wear-leveling. The results are used to deduce required wear-leveling schemes, which are implemented and deployed back to the simulator afterwards. This end-to-end analysis evaluates the implemented wear-leveling algorithm, respecting all interactions with the underlying operating system and the executing hardware. The results can be used to determine the quality of the wear-leveling algorithm and to deduce further required wear-leveling schemes, for instance an algorithm targeting a finer granularity. A comparison to a high-level analysis, which does not require a real implementation of the wear-leveling algorithm at all, points out there is a lack of preciseness in the evaluation, leading to different results.

## V. Performance Improvement and Alternative Simulators

Due to the need of a full-system simulator in the aforementioned setup, the performance faces a certain drawback. This can be mainly observed by two effects. First, the simulations are very time consuming and computations are generally slow. For example, an application running for one second on a native 2GHz machine requires multiple hours of simulation time. This is caused by the fact that the entire CPU and machine is simulated in software as a Linux application. Second, the trace volume for the memory trace file is large. NVMain2.0 by default writes the trace file as a human readable text file, including precise information for every single memory access. The volume of this trace file limits the speed of subsequent analyses, as the file has to be loaded and processed. Furthermore, with a slow main memory / disk configuration on the simulation host, the process of writing the trace file could also limit the simulation speed itself.

To overcome these performance impacts, several improvements can be applied to the simulation setup. In this section, first the possible improvements for the simulation speed are discussed in Section V-A. Afterwards, the potential to reduce the volume of the memory traces is presented in Section V-B.

### A. Simulation Speed Improvements

The main reason for the low performance is the gap between a native system and the simulation of functional units on a cycle level by gem5. Every functional unit is simulated by a software component, including a precise model to simulate the appropriate behavior of the functional unit. The composition of the functional units to the CPU and further peripherals leads to the execution of many instructions for the simulation of a single CPU cycle. Thus, many CPU cycles on the host machine have to be spent to simulate a single CPU cycle of the target.

As the accuracy of the simulation has to be maintained, the only possibility to speed up the simulation is to avoid the execution of the aforementioned instructions at each simulation cycle. One possibility is to not simulate the target hardware, but to rather run the application directly on the native target platform. This only requires an additional mechanism to extract the detailed information, which are usually gathered through the simulator. In our simulation setup, the type and target of memory accesses have to be extracted during the software execution. The further techniques presented in this paper can be applied based on this information, regardless of the source and the acquiring method of the information. Bao et al. [3] propose a hardware setup using an FPGA board, which is plugged in the DRAM slot of a machine and snoops the memory accesses to the DRAM module, to collect the memory access information of a standard computer. Furthermore, similar simulators are being proposed, which make use of on-chip FPGAs to also snoop and collect the memory access information [11], [14]. Either way, the concept to snoop memory accesses of a real device is significantly faster than a device simulation and provides the possibility to trace memory accesses of the CPU for further analysis.

However, the requirement of an on-chip FPGA or a complex, specialized hardware setup limits the choice of target platforms and implies a strong dependency on the platform specific memory subsystem. As an alternative, we consider to use the CPU integrated hypervisor mode to extract and process the memory access information. Traditionally, the hypervisor is used to supervise, preempt and switch the execution of multiple operating system. To achieve this, the hypervisor mode is equipped with advanced privileges to control the operating system. As part of these, the hypervisor can setup a memory configuration for the execution of the underlying operating system. This allows the hypervisor to configure the memory in a way that every memory access is trapped to the hypervisor and thus can be processed. The wide availability of hypervisor extensions makes this approach easily applicable. Furthermore, memory accesses are considered on the software execution level and do not require precise knowledge about the underlying memory subsystem. As a drawback, we expect this approach to face a higher overhead than a hardware (FPGA) based solution. Nevertheless, it may still be a significant improvement compared to the full-system simulation.

### B. Trace Volume Improvements

The performance of our simulation setup is also reduced by the large size of the resulting memory trace files. For a simulation of a program, which runs for some seconds on the native platform, the size of the memory trace reaches several gigabytes easily. Beside the fact that storing multiple gigabytes usually requires some storage management, the processing of the trace files is time consuming. To apply the filters and further aggregation mechanisms on the traced memory accesses, the entire trace file has to be loaded and processed, which consumes additional time. We are developing several solutions to reduce the impact of the large trace file volume.

One approach is to directly apply the filters and aggregations during the simulation process, which omits the steps of writing out the trace file and reading it in again for the purpose of processing. The drawback of this approach is the loss of information due to the early applied filters and aggregations. The targets of the analysis have to be determined before running the simulation or the simulation has to be repeated for every analysis target. A considerable hybrid solution could be to perform the analysis during the simulation, but still write out the conventional trace file. While this does not overcome the issue of storing large trace files, it improves the speed of the first analysis and allows arbitrary analyses later on.

Another approach is to reduce the volume of the trace file by applying lossless and lossy compression mechanisms. By default, the NVMain2.0 trace files contain human readable information for every memory access, which is space consuming without compression. One straightforward mechanism to reduce the size is to switch to another binary encoding of the trace file, for instance, one that stores a cache line in 64 bytes instead of 130 characters. Further looseless compression (e.g. delta encoding for the clock cycle) can be applied on this basis afterwards. Also lossy compression can be applied by filtering out information in the trace file that is not used for any analysis. For instance, the CPU core of the memory access can be omitted for a single core simulation, as well as the old memory content for read operations.

In a test, we changed the trace-writer to a binary format with variable length for every data field. Additionally, the controller cycle information and the access address are delta encoded. This ensures that the trace file is still readable in a subsequent order and every single access can be further processed. The original trace file size (human readable) of 244 MB can be reduced by this simple encoding to 165 MB, which is an improvement of 32%. Note that a larger improvement is possible, e.g., gzip [6] reduces the size by $91,7\%$. However, the proposed format allows trace rewriting and processing with extreme low overhead for decompression.

## VI. Conclusion

In this paper, we propose a simulation setup for the analysis of application's memory access behavior, which takes the underlying operating system and hardware into account. This allows to analyze algorithms, which target the memory behavior (e.g. wear-leveling algorithms), with respect to the modified write behavior, for instance due to operating system interactions and a changed caching behavior. As a demonstration of our setup, we perform a case study on the development of a specialized software-only wear-leveling algorithm. The simulation results are used to specify the target of the algorithm as well as to evaluate the algorithm in the full setup.

Since the performance of the simulation setup was identified as a significant drawback during our experiments, we propose several modifications to increase the simulation speed as well as to decrease the produced trace volume size. This also includes alternative simulation environments, which are hardware-aided.

Our simulation setup is available at [2]. We appreciate not only software based solutions to be analyzed by this framework, but also functional simulations of proposed hardware controller to be integrated and evaluated.

## Acknowledgement

This paper is supported in parts by the German Research Foundation (DFG) Project OneMemory (CH 985/13-1).

## References

[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] Nvm simulator. https://github.com/tu-dortmund-ls12-rt/NVMSimulator.

[3] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei 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.

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

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[6] L Peter Deutsch. Gzip file format specification version 4.3. 1996.

[7] Jianbo Dong, Lei Zhang, Yinhe Han, Ying Wang, and Xiaowei Li. Wear rate leveling: Lifetime enhancement of pram with endurance variation. In *Proceedings of the 48th Design Automation Conference*, pages 972–977. ACM, 2011.

[8] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):994–1007, 2012.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[10] 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.

[11] Taemin Lee, Dongki Kim, Hyunsun Park, Sungjoo Yoo, and Sunggu Lee. Fpga-based prototyping systems for emerging memory technologies. In *2014 25nd IEEE International Symposium on Rapid System Prototyping*, pages 115–120. IEEE, 2014.

[12] Wei Li, Ziqi Shuai, Chun Jason Xue, Mengting Yuan, and Qingan 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.

[13] 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.

[14] Yu Omori and Keiji Kimura. Performance evaluation on nvmm emulator employing fine-grain delay injection.

[15] 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.

[16] 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.

[17] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference*, pages 37–49. ACM, 2015.

[18] Jibin Wang and Bohan Wang. Pcmsim: A hybrid memory system simulator for the cloud storage. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 81–86. IEEE, 2017.

[19] M. Zhao, L. Shi, C. Yang, and C. J. Xue. Leveling to the last mile: Near-zero-cost bit level wear leveling for pcm-based main memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 16–21, Oct 2014.

[20] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter A Boncz. Super-scalar ram-cpu cache compression. In *Icde*, volume 6, page 59, 2006.