Towards Non-Volatile Memory Wear-Leveling In (Timing)-Critical Systems

Christian Hakert, Kay Heider, Junjie Shi, Jian-Jia Chen
TU Dortmund University, Germany

***Abstract*—Addressing the lifetime challenges of emerging non-volatile memory through wear-leveling has been widely discussed in the literature, with many approaches focusing on practical implementations. Most efforts prioritize extending system lifetime by minimizing the overhead of wear-leveling techniques. However, in safety-critical and time-sensitive systems, the emphasis shifts from minimizing overheads to ensuring system reliability. In such systems, guaranteeing that memory will not degrade or wear out under any conditions throughout the targeted lifetime becomes the primary objective of wear-leveling.**

In this paper, we address this previously overlooked challenge by ensuring lifetime feasibility for real-time systems. To this end, we extend the classical real-time task model by introducing a new parameter: worst-case wear-out. Using this extended task model, we develop a wear-leveling scheme that ensures that memory cells do not exceed their endurance limits during the lifetime. Additionally, we compute the required number of memory replicas necessary to meet the system's lifetime requirements.

***Index Terms*—Non-Volatile Memory, Real-Time Systems, Wear-Leveling**

I. Introduction

Emerging non-volatile memory (NVM) technologies, such as phase change memory (PCM) and ferroelectric RAM (FeRAM), are increasingly being integrated into off-the-shelf microcontrollers, including the Stellar MCUs from STMicroelectronics and the MSP430FRXXX series from Texas Instruments. Especially in the design of efficient timing and safety critical systems, these emerging technologies provide favorable properties, such as fast hibernation and large power saving potential [4]. A key challenge for NVM-based systems lies in the limited endurance of these memory technologies, which can be as low as 10^8 write cycles [1]. Wear-leveling is a commonly used technique to address this issue by redistributing memory accesses to ensure that memory cells experience uniform wear. While this technique effectively prolongs the memory's lifespan, it introduces time overhead and must be performed in a timely manner to prevent premature wear-out of individual cells. In the context of critical real-time systems where critical tasks are deployed to emerging memory due to the aforementioned advantages, it is essential not only to ensure timing feasibility, where all jobs meet their deadlines, but also lifetime feasibility, meaning the system can operate within its targeted lifetime without exhausting the endurance of any memory cell.

To achieve both timing and lifetime feasibility in wear-leveling, we define the memory behavior of a task using a worst-case wear-out (WCWO) metric for each job, analogous to the worst-case execution time (WCET). We introduce *migration operations*, which preemptively relocate a task's memory to a different physical location before the execution of a new job. The overhead incurred by these migration operations is incorporated as an additional WCET component for the task. The selection of migration locations is determined by a wear-leveling algorithm that employs two key strategies. First, it ensures that different tasks are cyclically allocated to memory locations, achieving a task-specific average wear-out for each memory block. Second, it incorporates additional memory replicas to ensure that each memory cell can endure until the system's lifetime is reached. The required number of replicas can be calculated during system deployment by analyzing the task-specific average wear-out. This approach guarantees that migration operations will not compromise the system's timing feasibility, while ensuring that memory locations are uniformly and progressively worn out, resulting in a lifetime-feasible system.

To the best of our knowledge, we present the first approach that integrates wear-leveling to ensure both lifetime and timing feasibility in critical real-time systems. Our key contributions are summarized as follows:

- We introduce an extended task model that incorporates the WCWO for each job, analogous to the WCET.
- We propose a wear-leveling strategy that guarantees both lifetime and timing feasibility, while optimizing the required number of memory replicas by leveraging task set-specific properties.
- We provide a comprehensive evaluation of the proposed wear-leveling scheme through both simulation-based and measurement-based methods.

II. Related Work

While wear-leveling of non-volatile memories is covered by a wide range of practical approaches in the literature, where the focus is put to different technology assumptions and different specialized approaches, considering the lifetime issue from a systematic perspective on giving guarantees and guaranteeing the lifetime feasibility of a system is widely lacking. General approaches aim for efficient wear-leveling and the reduction of overheads in general [6], [7]. Hu et al. especially focus on the efficiency of software based wear-leveling mechanisms [6]. Trading off between software-based wear-leveling and hardware-supported wear-leveling, especially with a focus on the overheads, is a widely discussed topic. Hakert et al. introduce a comprehensive methodology for software-based wear-leveling [5]. Specific approaches target to

aim for better efficiency by exploiting the stack as a special software construct [11].

An approach for formalization of wear-leveling is given by Onodera and Shibuya [10], where the lacking formalism of wear-leveling in the literature is picked up and a rigorous approach for theoretical lifetime analysis is followed. However, the authors do not lay a dedicated focus on real-time systems and the timing feasibility. Lee et al. focus on wear-leveling in the real-time operating-system FreeRTOS, where they propose a safe online stack migration scheme [8]. Although this is a crucially important practical implementation aspect to wear-leveling in real-time systems, the consideration about lifetime feasibility, as proposed by this paper, are not picked up.

III. System Model

This section introduces the memory lifetime model, followed by an explanation of the lifetime requirements for real-time systems.

A. Memory Lifetime Model

We consider a system equipped with main memory implemented using a NVM technology, which is characterized by limited cell endurance. Specifically, each memory cell has a finite number of write cycles it can endure before becoming unreliable. In the absence of wear-leveling or other mechanisms, the system is assumed to become not operable once the first memory cell fails. Therefore, we define a system-wide parameter, i.e., E, representing the endurance of a single memory cell. The system's lifetime is defined as the period during which no memory cell exceeds E write operations. The goal of lifetime extension is to prolong the system's operation until this threshold is reached.

In addition, the system is assumed to include a small amount of conventional memory, that is used to store centralized state information required for the lifetime extension mechanism.

B. Characterization of Lifetime Requirements

To take memory lifetime into considerations for a real-time system, we extend the classic periodic real-time task model by introducing an additional parameter, the *worst-case wear-out* (*WCWO*). Accordingly, we denote a periodic task τ_i by $\tau_i = (\phi_i, C_i, T_i, D_i, \mathit{WCWO}_i)$, where ϕ_i is the phase, C_i is the (worst-case) execution time, T_i is the period, D_i is the relative deadline and WCWO_i is the worst-case wear-out. In this work, we limit our focus to strictly periodic tasks. The worst-case wear-out denotes the maximum number of write operations to any memory location during the execution of a single job. Similar to the worst-case execution time, this parameter can be determined through measurement-based approaches.

For measurement-based estimation, memory instrumentation tools such as Dynamorio [3] or Valgrind [9] can be employed to track and record individual memory accesses to specific locations during task execution. Summing these accesses per location provides an estimate of the worst-case wear-out, which may be improved by applying a safety margin.

Alternatively, as with execution time, the worst-case wear-out can be safely upper-bounded using static program analysis. In this approach, the control flow graph of the task is explored, and predictions are made regarding the target address of each memory access. If precise predictions cannot be made safely, memory accesses are pessimistically assumed to potentially target any location associated with the task. This allows the determination of the worst-case path through the control flow graph, yielding a safe upper bound for the worst-case wear-out. Tools like BAP [2] can assist in static program analysis by providing more accurate predictions of memory access targets, thus tightening the upper bound.

We assume the main memory to consist of n fragments, denoted as $\mathit{MF} = (\mathit{mf}_0, ..., \mathit{mf}_{n-1})$, to which any task can be freely mapped. Additionally, we assume that tasks only share read-only memory[1]. A task-to-memory mapping function $\mathit{TM}\colon T \times L \rightarrow \mathit{MF}$ can be employed, where $T = (\tau_0, ..., \tau_{m-1})$ represents the set of m tasks in the system, and $L = (t_0, ..., t_{\ell-1})$ denotes the number of system ticks until the system's lifetime expiry. This mapping can change at any tick; however, any changes affect only the next released job. This means, the actual migration of a task is executed at the next release after the corresponding change of the mapping. Additionally, we denote $J_{k,j}$ as the j-th job of task τ_k that is released at tick $R(J_{k,j}) = \phi_k + T_k \cdot (j-1)$.

For notational brevity, let

$$G_{\gamma,i} = \begin{cases} \mathit{WCWO}_k, & \text{if } \mathit{TM}(\tau_k, t_\gamma) = \mathit{mf}_i \wedge \gamma = R(J_{k,j}) \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Then, the accumulated worst-case wear-out AWO_i for each memory fragment i can be computed as follows:

$$\mathit{AWO}_i = \sum_{\gamma=0}^{\ell-1} G_{\gamma,i} \tag{2}$$

Following the assumption that the system becomes unusable once the first memory cell wears out, the memory fragment with the maximum accumulated wear-out is of critical interest and defines the global memory wear-out (*GWO*):

$$\mathit{GWO} = \max(\mathit{AWO}_0, ..., \mathit{AWO}_{n-1}) \tag{3}$$

We denote the system to be *lifetime feasible* for a given lifetime ℓ when $\mathit{GWO} \leq E$. The correct choice of *TM* is the key parameter to achieve lifetime feasibility for a targeted system lifetime ℓ. The process for deriving a suitable *TM* is explained in the following section.

IV. Wear-Leveling

The behavior of tasks in a system can significantly impact memory endurance, potentially causing memory cells to wear out prematurely, even before the system reaches its targeted lifetime. For example, consider a phase-change memory (PCM)-based system with a cell endurance of 10^8 write cycles

[1]This can be achieved straightforwardly by following the principle of queue-based communication, thereby avoiding written shared memory.

[1], and a task with a period of $10\,\mathrm{ms}$ that performs a worst-case of 10 memory write accesses per job. In this scenario, the maximum feasible lifetime without wear-leveling, where *TM* remains constant throughout the system's lifetime, would be approximately one day, i.e., $\frac{10^8}{10 \cdot 100 \cdot 60 \cdot 60 \cdot 24} \approx 1$ day. Such a lifetime falls far short of the intended lifespan for many critical systems. Consequently, the task-to-memory mapping function *TM* must be adjusted to ensure the system remains lifetime feasible for a given target lifetime.

In this work, we propose a wear-leveling scheme to derive a suitable *TM* by utilizing the following strategies: 1) spare memory is employed to remap tasks to unworn memory regions ($n \gg m$), and 2) task assignments are shuffled in such a way that memory fragments experience uniform worst-case wear-out. For brevity and simplicity, we assume that the targeted lifetime, ℓ, is an integer multiple of the hyper-period of the task set, denoted as *HP*, and that ℓ is also a common integer multiple of the worst-case wear-out across all tasks.

Our wear-leveling scheme modifies the task-to-memory mapping at the frequency of the hyper-period, ensuring that each task releases the same number of jobs and at least one job between two mapping changes. At each change, all tasks are relocated to the next memory replica in a wraparound fashion. Within the replicas, tasks are also mapped with an increasing offset, maintaining the wraparound semantic. Consequently, when r denotes the number of memory replicas, after $r \cdot m$ mapping changes, each task will have been mapped to each memory fragment exactly once for the duration of one hyper-period. Figure 1 illustrates this remapping process for two tasks, each with two memory segments per replica, and three replicas. When the system lifetime ℓ is an integer multiple of $HP \cdot r \cdot m$, each memory fragment experiences the same number of executed jobs for each task over the entire system lifetime. It should be noted that, due to larger hyper-periods, the lifetime requirement of the system may be limited to an absolutely high scale. The wear-leveling algorithm potentially can be modified in order to operate on a higher frequency than the hyper-period. This, however, will not ensure the same number of job invocations between task migrations and potentially has to include a more pessimistic estimation on memory segment ages. A tight integration with the scheduling algorithm can potentially help to make these estimations less pessimistic.

We define $n = r \cdot m$, where r represents the number of memory replicas and m is the number of memory fragments required by the task set. Then, the wear-leveling strategy can be described by:

$$TM(\tau_i, t_j) = \begin{cases} \Big(\big((TM(\tau_i, t_{j-1}) + 1)\%m\big) + m\Big)\%n, & \text{if } j\%HP = 0 \\ TM(\tau_i, t_{j-1}), & \text{otherwise} \end{cases} \tag{4}$$

where % is the mod operation.

This strategy ensures that tasks are systematically remapped to different memory fragments, thus distributing wear uniformly across the available memory space and extending the overall system lifetime.

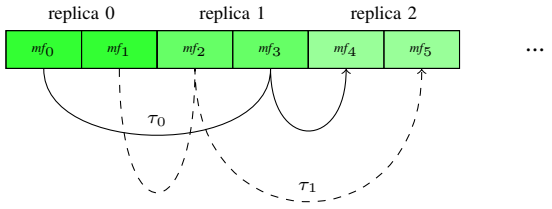

Fig. 1. Wear-Leveling Scheme Illustration

For each task τ_i, the worst-case wear-out for one hyper-period is given by $\frac{HP}{T_i} \cdot WCWO_i$. Under the described wear-leveling strategy, each memory fragment is subject to a proportional share of task execution, and the wear-out of each memory fragment can be expressed as:

$$AWO_i = \frac{1}{r \cdot m} \cdot \frac{\ell}{HP} \cdot \sum_{j=0}^{m-1} \frac{HP}{T_j} \cdot WCWO_j + \frac{\ell}{HP \cdot r} \tag{5}$$

$$= \frac{\ell}{r \cdot m} \cdot \sum_{j=0}^{m-1} \frac{1}{T_j} \cdot WCWO_j + \frac{\ell}{HP \cdot r} \tag{6}$$

In these equations, $\frac{\ell}{HP \cdot r}$ accounts for the memory write overhead required to migrate each task fragment to the next location during a mapping change. The normalized worst-case wear-out of a task τ_i is defined as $NEW_i = \frac{1}{T_i} \cdot WCWO_i$. Under perfect wear-leveling, each memory fragment should experience the same average normalized wear-out across all tasks, denoted as $MNEW = \frac{1}{m} \cdot \sum_{i=0}^{m} NEW_i$. As shown in Equation (6), the proposed wear-leveling strategy achieves this balanced distribution of wear across all memory fragments. Since all fragments are subject to the same *MNEW* after wear-leveling, the endurance limit E applies equally to all fragments:

$$E = \frac{\ell}{r} \cdot MNEW + \frac{\ell}{HP \cdot r} \tag{7}$$

$$\Leftrightarrow r = \frac{\ell}{E} \cdot (MNEW + \frac{1}{HP}) \tag{8}$$

Equation (8) provides the formula to calculate the required number of replicas r to achieve the targeted system lifetime ℓ, considering the cell endurance E, the task set-specific mean normalized wear-out *MNEW*, and the hyper-period *HP*.

V. Migration Overhead

In the previously described wear-leveling algorithm, the overhead associated with migrating a task to a different memory fragment is included in the total number of memory writes. However, this migration process also incurs execution time overhead, which must be considered to ensure the system's timing feasibility. This overhead can be safely upper-bounded and added to the WCET of each task. Since migrations occur only once per hyper-period, and each task executes at least one job during this period, the overhead is adequately accounted for. The upper bound on migration overhead is determined by the worst-case execution time required to copy the task-related

data to a new memory location, which depends on the size of the memory fragments and hardware-specific parameters.

We evaluated the migration overhead in FreeRTOS by measuring the time required to copy the task control block (TCB) data structure and the stack memory associated with a task. To correctly migrate a task's stack to a new memory location, it is necessary to adjust any materialized addresses on the stack that point to memory within the stack itself. Specifically, if a value on the stack references an address in the old stack memory, this value must be updated to reflect the new stack location.

In addition to migrating the stack, the task's TCB must also be copied. In FreeRTOS, the TCBs of all configured tasks are managed in doubly linked lists based on their current state. For instance, the TCBs of tasks that are ready for execution are referenced in a global ready list. Whenever a TCB is migrated to a new memory location, the corresponding list that owns the TCB must be updated to reflect the new location.

To synchronize task migration with task execution, we introduce a migration flag in the TCB, indicating a pending migration. When the task is released for its subsequent job, the migration process is triggered, ensuring seamless task execution in the new memory location.

We experimentally evaluate the time required to migrate a task on an ESP32-S3 microcontroller, configured with a single core running at $240\,\mathrm{MHz}$. The FreeRTOS port provided by the manufacturer is utilized for this evaluation. The system tick period is set to $1\,\mathrm{ms}$. A single task is created with a period of 3 ticks and an execution time of ≈ 1 tick. The stack sizes are varied between $2\,\mathrm{KiB}$ and $64\,\mathrm{KiB}$, while the TCB size remains fixed at $348\,\mathrm{B}$. In total, the task is migrated 10 times, and the maximum migration time is recorded alongside the maximum task runtime.

Figure 2 presents the maximum migration time (red crosses) and the maximum runtime of the task (green triangles) for each memory configuration. Furthermore, a linear regression model of the migration time is shown as a blue line. The x-axis represents the total amount of memory copied during each migration, while the y-axis indicates the elapsed time.

The results demonstrate that migration time increases linearly with the amount of copied memory, starting at $\approx 50\,\mu\mathrm{s}$ for $2396\,\mathrm{B}$, and up to $\approx 1250\,\mu\mathrm{s}$ for $65\,884\,\mathrm{B}$. The data reveals that migration overhead becomes significant, especially for larger stack sizes. In particular, at $53\,596\,\mathrm{B}$, the migration time exceeds the runtime of the actual task. This suggests the conclusion that the overhead of the proposed wear-leveling method carefully has to be considered specifically for an application set and hardware platform. Under some applications, only minimal overhead has to be sacrificed for proper lifetime extension, for other application, the majority of potential system utilization may have to be sacrificed.

To provide a measurement-based approximation of the WCET for the migration process, we fit a linear regression model to the recorded data. A safety margin of 10% is applied to both the slope and intercept to obtain a conservative upper bound for the migration process across all observed data points. The migration overhead in microseconds, based on the amount of memory copied s, is described by the following formula:

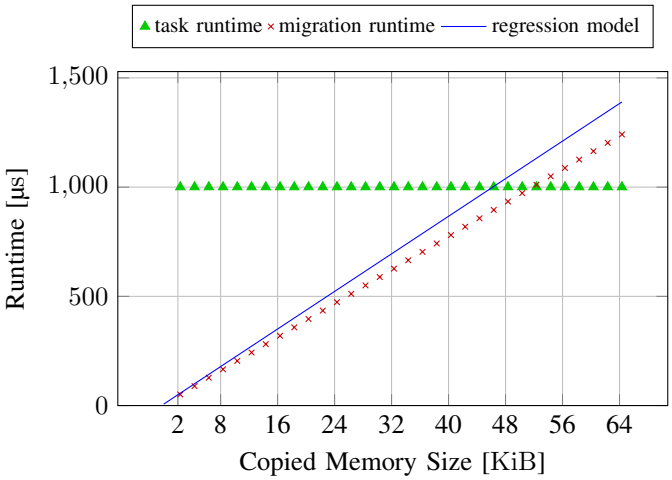

Fig. 2. Migration Time Overhead

$$O = 0.021 \cdot s + 6.452 \tag{9}$$

Using this model, the WCET of a task τ_i, including the migration overhead, can be expressed as $C_i' = C_i + O$. Please note that this approach is conservative, as the migration occurs only once per hyper-period, rather than for every job released by the task.

VI. Simulation Study

To provide intuition for the previously introduced wear-leveling concept with memory replicas, we conduct a simulation study and illustrate the resulting parameters for simulated task sets. For an exploratory analysis, we randomly generate task sets where the periods are uniformly distributed between 10 and 15 ticks, and the WCET is between 1 and a maximum of $\frac{1}{5}$th of the period. Additionally, the worst-case wear-out for each task is randomly sampled between 5 and 10. We generate tasks such that the total system utilization remains below 100%. We assume a cell endurance of 10^8 write accesses and a targeted system lifetime of 10 years without interruption.

For each task set generated according to this procedure, we plot a single point in Figure 3, where the x-axis is the mean normalized wear-out (MNEW) and the y-axis indicates the required number of replica sets, computed by Equation (8). The MNEW values for the generated task sets range between approximately 0.5 and 0.8, and the MNEW appears to be linearly related to the required number of replicas. However, deviations from this linear relationship occur due to two factors: 1) the length of the hyper-period, and 2) rounding to the nearest whole number of replicas.

Next, we provide intuition for the potential benefits of the proposed wear-leveling scheme by comparing it to a baseline approach in which no active wear-leveling is applied, and only a replica strategy is used. In this baseline, the maximum WCWO for each task invocation is considered as an upper

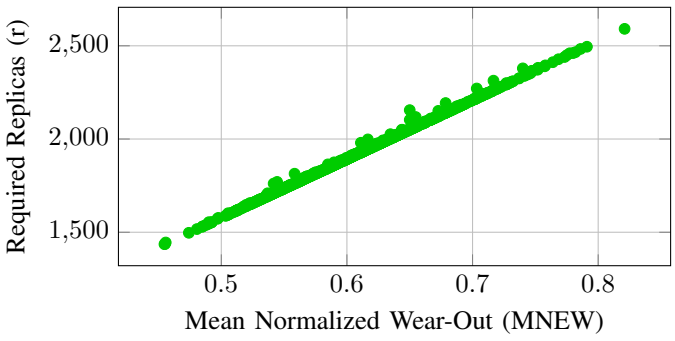

Fig. 3. Required Replicas for Sampled Tasksets

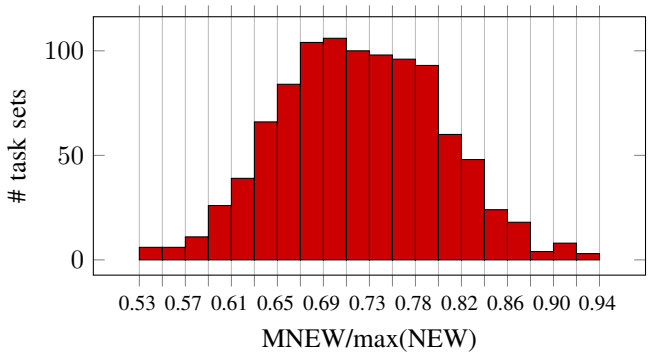

Fig. 4. Potential of Wear-Leveling

bound. To illustrate the difference between the proposed scheme and the baseline, we present the relation between the MNEW and the maximum NEW of each task as a histogram across the generated task sets in Figure 4. The results show that the MNEW can be up to $\approx 50\%$ lower than the maximum NEW in the generated task sets, highlighting the potential benefit of the wear-leveling scheme. Additionally, we simulate the invocation of tasks under the proposed wear-leveling scheme throughout the system's lifetime and record the memory aging for each memory segment. Due to computational limitations, we simulate a system lifetime of $\approx 1.4 \cdot 10^8$ ticks, instead of the full 10-year lifetime. This simulation requires the use of two replica sets for the generated task set. In Figure 5, we plot the maximum memory age at each tick (blue line) alongside the memory's endurance limit (red line) on the y-axis, with the system ticks on the x-axis. The results indicate that the maximum memory age does not exceed the memory's lifetime limit in the example, demonstrating the effectiveness of the proposed wear-leveling approach.

VII. Conclusion

In this paper, we address the overlooked issue of NVM wear-leveling in critical, timing-constrained systems by proposing a wear-leveling scheme that guarantees the lifetime feasibility of a real-time system for a given targeted system lifetime. To achieve this, we extend the classical real-time task model to include worst-case wear-out and base our analysis on this extension. By considering the targeted system lifetime,

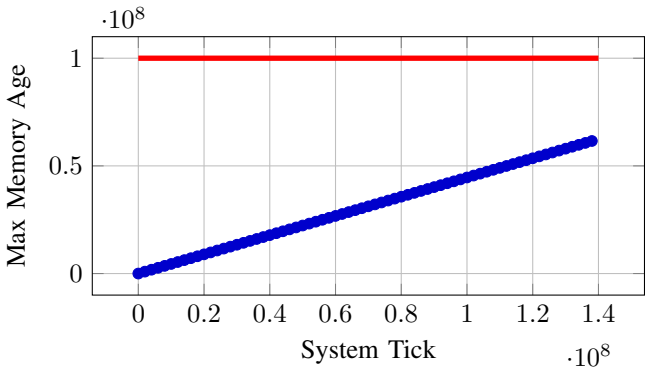

Fig. 5. Memory Wear-Out Simulation

we ensure lifetime feasibility through the introduction of a predictable number of memory replicas. Additionally, we conducted a practical evaluation to estimate the overhead of migration operations, which are factored into the task's WCET. Finally, we performed a simulation study to determine the required number of memory replicas, demonstrating the effectiveness of the proposed scheme.

VIII. Acknowledgements

This work has received funding from the DFG Priority Program "Disruptive Memory Technologies" (SPP 2377) as part of the project "ARTS-NVM" (502308721) and "Memory Diplomat" (502384507). It was further supported by the DFG Project "One-Memory" (405422836). This result is part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170).

References

[1] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging nvm: A survey on architectural integration and research challenges. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(2):1–32, 2017.

[2] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 463–469. Springer, 2011.

[3] Hyoun Kyu Cho, Tipp Moseley, Derek Bruening, and Scott A Mahlke. Instant profiling: Instrumentation sampling for profiling datacenter applications. 2013.

[4] Mario Günzel, Christian Hakert, Kuan-Hsun Chen, and Jian-Jia Chen. Heart: H ybrid memory and e nergy-a ware r eal-t ime scheduling for multi-processor systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–23, 2021.

[5] Christian Hakert, Kuan-Hsun Chen, Horst Schirmeier, Lars Bauer, Paul R Genssler, Georg von der Brüggen, Hussam Amrouch, Jörg Henkel, and Jian-Jia Chen. Software-managed read and write wear-leveling for non-volatile main memory. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(1):1–24, 2022.

[6] Jingtong Hu, Mimi Xie, Chen Pan, Chun Jason Xue, Qingfeng Zhuge, and Edwin H-M Sha. Low overhead software wear leveling for hybrid pcm+ dram main memory on embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(4):654–663, 2014.

[7] Jiacheng Huang, Min Peng, Libing Wu, Chun Jason Xue, and Qingan Li. Lamina: low overhead wear leveling for nvm with bounded tail. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 377–382. IEEE, 2022.

[8] Jeongmin Lee, Moonseok Jang, Kexin Wang, Inyeong Song, Hyeonggyu Jeong, Jinwoo Jeong, Yong Ho Song, and Jungwook Choi. Improving nvm lifetime using task stack migration on low-end mcu-based devices. *IEEE Access*, 10:125319–125333, 2022.

[9] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[10] Taku Onodera and Tetsuo Shibuya. Wear leveling revisited. In *31st International Symposium on Algorithms and Computation (ISAAC 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2020.

[11] Jifeng Wu, Wei Li, Libing Wu, Mengting Yuan, Chun Jason Xue, Jingling Xue, and Qingan Li. Effective stack wear leveling for nvm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(10):3250–3263, 2023.