Stack Usage Analysis for Efficient Wear Leveling in Non-Volatile Main Memory Systems

Christian Hakert, Mikail Yayla, Kuan-Hsun Chen,
Georg von der Brüggen, Jian-Jia Chen,
Sebastian Buschjäger, Katharina Morik,
Paul R. Genssler, Lars Bauer, Hussam Amrouchm, Jörg Henkel
Department of Computer Science, TU Dortmund, Germany
https://ls12-www.cs.tu-dortmund.de/

Citation: https://doi.org/10.1109/MLCAD48534.2019.9142113

BibTeX:

```
@inproceedings { mlcad2019stackanalysis,
  author = {Hakert, Christian and Yayla, Mikail and Chen, Kuan-Hsun and Br\"uggen, Georg von der
  and Chen, Jian-Jia and Buschj\"ager, Sebastian and Morik, Katharina and Genssler, Paul R.
  and Bauer, Lars and Amrouch, Hussam and Henkel, J\"org},
  title = {Stack Usage Analysis for Efficient Wear Leveling in Non-Volatile Main Memory Systems},
  booktitle = {1st ACM/IEEE Workshop on Machine Learning for CAD (MLCAD) },
  year = {2019},
  address = {Alberta, Canada},
  confidential = {n},
}
```

Stack Usage Analysis for Efficient Wear Leveling in Non-Volatile Main Memory Systems

Christian Hakert, Dortmund, DAES
Mikail Yayla, Dortmund, DAES
Kuan-Hsun Chen, Dortmund, DAES
Georg von der Brüggen, Dortmund, DAES
Jian-Jia Chen, Dortmund, DAES
Sebastian Buschjäger, Dortmund, AI Group
Katharina Morik, Dortmund, AI Group
Paul R. Genssler, KIT, CES
Lars Bauer, KIT, CES
Hussam Amrouch, KIT, CES
Jörg Henkel, KIT, CES

***Abstract*—Emerging non-volatile memory (NVM) technologies, such as Phase Change Memory (PCM), have been considered as a replacement for DRAM and storage due to their low power consumption, fast access speed, and low unit cost. Even so, some NVMs have a significantly lower write endurance and hence in-memory wear leveling is an important requirement for practical applicability. Since writes to the stack often target a small and dense memory region, generic, coarse-grained wear-leveling mechanisms (e.g. virtual memory page remapping) are not sufficient. An alternative solution is to relocate the stack memory regularly, which involves copying of the stack content. As the stack content changes in size during the execution of an application, the copy overhead can be significantly mitigated by performing the relocation when the stack size is small.**

In this paper, we investigate two approaches to determine points in time when the stack is small. First, we analyze the possibility to fit simple machine-learning models to the stack usage function. Precise predictions of this function enable the identification of the minimum stack size during execution. In our evaluation, the tested models provide accurate estimates of the future stack usage function for a subset of common applications.

As a second approach, we analyze applications a priori and determine potential optimal points to perform relocation in the instruction stream. In detail, we deploy the application in an analysis environment, which determines a rating for each executed instruction. Based on this rating, we apply a genetic algorithm to identify the best points in the instruction stream to perform the stack relocation. This approach allows to save up to 85% of the write overhead for wear-leveling in our experiments.

***Index Terms*—Non-Volatile Memory, Wear-Leveling, Genetic Algorithm, Decision-Tree**

I. Introduction

Recently emerging non-volatile main-memory technologies result in a need for in-memory wear-leveling. For instance, while DRAM endures for at least 10 years under a given usage pattern, phase change memory (PCM) would wear out within 5 minutes under the same pattern [4]. Hence, several wear-leveling techniques have been studied in literature, exploring a variety of algorithms, hardware supports, and software supports. One concept is to realize the wear-leveling without any special hardware support to save the required chip-space. It balances the wear-levels by using the virtual memory feature of the memory management unit (MMU) and maintaining a mapping of the entire main memory. This approach can be classified as coarse-grained since its precision is restricted to the virtual memory granularity of typically 4 kB. Hence, a highly non-uniform memory usage within the virtual memory pages cannot be handled properly by this approach.

Analyses of the memory usage of typical applications show that the stack segment is responsible for the majority of non-uniform memory usage within virtual memory pages. Thus, a wear-leveling technique that specifically targets the application's stack has considerable advantages. One approach is to copy the stack memory within a fixed memory region, i.e., multiple virtual memory pages, in a circular manner, distributing the non-uniform memory usage over this region. The main overhead of this method is caused by copying the current stack memory. The size of an application's stack memory is controlled by the positioning of the stack pointer.

To minimize the overhead of the stack relocation mechanism, it is necessary to perform the stack relocation when the applications stack is small. As the stack usage cannot be deduced manually in most cases, advanced analysis is required to determine the points in time when the stack is small and aid the wear-leveling with this information. We present two approaches to identify these time points in this paper. One is to fit specialized machine learning models to the stack usage during the application execution and to predict the future stack usage based on these models. Defining such a specialized model requires precise knowledge about possibly recurring patterns in the stack usage of typical applications. The second approach is to analyze the stack usage of a concrete application prior to the application execution. As this does not require a specialized model which has to fit the current stack usage, no further knowledge about stack usage patterns is required. The drawback of this approach is the need to perform the analysis for every application before its execution.

We perform a generic analysis of typical stack usage patterns in Section IV and determine three classes of stack functions for our benchmark applications, which can be used to classify the application. Second, we propose a genetic algorithm based method, which determines optimal stack relocation points a priori in Section V. Subsequently, we evaluate the usability of simple machine-learning models to predict a subset of classes of stack functions properly in Section VI.

978-1-7281-5758-0/19/$31.00 ©2019 IEEE

Such a model could later be deployed as a special hardware controller, which makes the analysis and annotation of the application needless as discussed in Section VII.

The novel contributions of this paper are:

1) An analysis framework to run an application and extract the size of the stack at any executed instruction.
2) The analysis of a set of benchmark applications to investigate the stack functions. Three classes of stack functions are deduced and the applications are categorized accordingly.
3) A generic analysis and annotation mechanism to allow efficient wear-leveling with an a priori analysis of an application.
4) An evaluation of simple machine-learning algorithms to determine how suitable they are to predict specific stack functions.

II. Related Work

The area of in-memory wear-leveling is targeted in the literature at different levels. Some approaches aim to perform the wear-leveling entirely in hardware [6], [12], others propose a combined hardware-software solution [10], and some works propose software-based wear-leveling [8], [9], [13]. Some of the software-based approaches consider specialized solutions for the stack memory in their work, i.e., a memory allocation is performed for the stack of each function call [8], [9]. They also mitigate the caused overhead for stack wear-leveling by only hooking in the wear-leveling algorithm for new memory requests. Thus, copying of old content is omitted. However, these approaches rely on the application's corporation to, for instance, perform a sufficient amount of function calls. The mechanism presented in this paper is more generic and hence requires different strategies to reduce the overhead.

III. Stack Usage Analysis Framework

In order to determine the stack sizes during the execution of an arbitrary application, a specialized framework is required. The size of the stack can be determined by reading out the stack pointer $sp_{current}$ and compare it to the stack pointer at the beginning of the application execution sp_{base}. The difference of these values determines the amount of valid stack memory. Thus, the main purpose of the analysis framework is to determine the stack pointer after the execution of certain instructions, i.e., the instructions which should be currently evaluated. This section gives a brief overview of how our analysis framework determines these stack pointers.

A. Ptrace Based Execution

The stack pointer is a CPU register, which can either be read and modified by the application itself or by the supervising operating system. As a monitoring of the stack pointer inside of the application requires, for instance, an instrumentation of the application code, this would influence the usage of the stack itself and lead to different results. Hence, we access the stack pointer from the supervising operating system. Linux provides an API, called *ptrace*, which can be used to debug

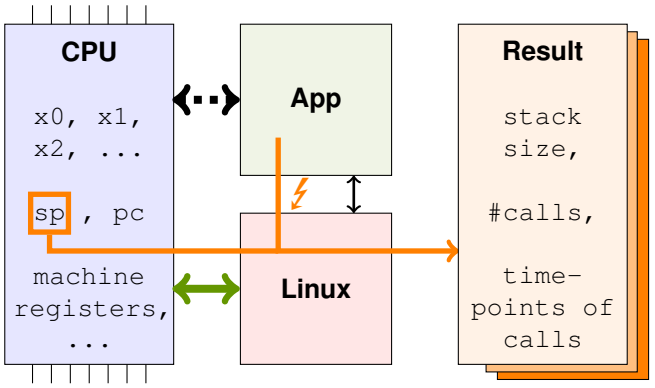

Fig. 1. Ptrace based analysis framework for instruction evaluation

an user level application, including the possibility to read the CPU registers at any point in time [2]. Ptrace can be configured to interrupt the application execution either after each instruction (single-step mode) or after a breakpoint instruction. The latter version requires an instrumentation of the code with breakpoint instructions, which, however, have no further influence on the application behavior. Whenever the breakpoint instruction is executed, it has to be replaced with the original instruction and the execution has to be continued. At every interrupt of the application, our framework reads out the CPU register set (including *sp*) with the ptrace API and determines the stack size after the execution of the current instruction.

B. Analysis Workflow

Whenever the analysis framework is called by, for instance, the training phase of a machine-learning model, the target application and an instruction address of interest are passed to the framework. The framework then initializes the ptrace environment and ensures that the application is interrupted after the execution of the instruction of interest by replacing it with a breakpoint instruction. The framework executes the application subsequently until termination or a timeout and returns a rating for the instruction of interest, including, e.g., the cumulative stack size over all executions of the instruction, the number of executions, and the points in time when the instruction is called. This workflow is illustrated in Figure 1.

This way of analysis is trivially parallelizable. The calling algorithm aims to always evaluate a set of instructions. To handle this, multiple instances of the analysis framework can be started, i.e., one for each instruction. We specifically target ARMv8 CPUs in our analysis [1]. For the execution of the analysis framework, we use an ARMv8 based Linux Server (2x Cavium Thunder X2), which offers 224 logical CPU threads.

IV. Stack Usage Pattern Analysis

To follow our first approach of fitting simple machine-learning models to the stack usage function, we first analyzed the stack functions of typical applications. We investigated several applications from the MiBench suite [7], but also self-written benchmarks. The stack function $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ maps

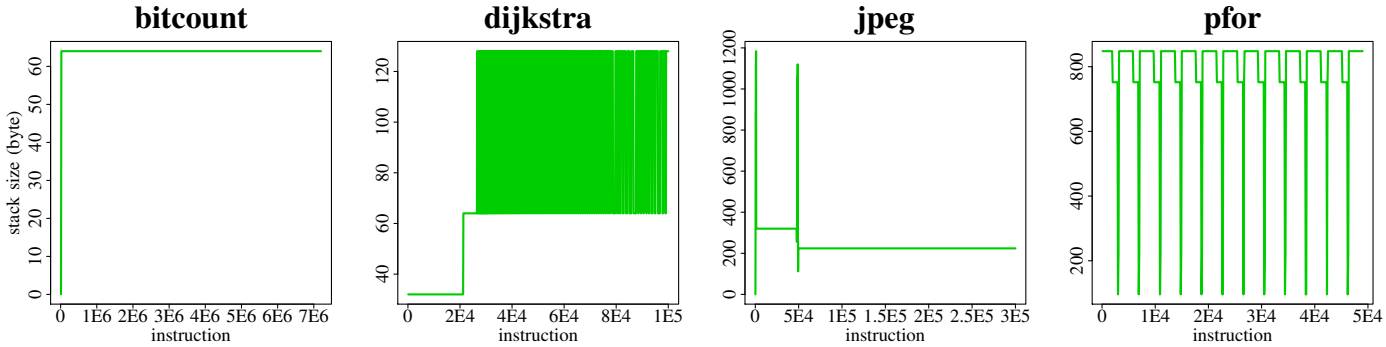

Fig. 2. The plots of four example applications, one out of every category (1. constant, 2. irregular, 3. constant, 4. periodic) in Table 2.

one instruction, identified by the clock cycle and thus called point in time, to its corresponding stack size during execution. We investigated the stack functions of various applications to abstract them to classes of stack functions. This allows us to decide which model should be used to model the stack function of an application. Figure 2 depicts the recorded stack functions of a small subset of applications and thus shows examples for different classes of stack functions. The y-axis shows the amount of used stack over each executed instruction (x-axis). As a result, we deduced three different classes of stack functions: 1) constant, 2) periodic, and 3) irregular. The categorization of the investigated applications is presented in Table I[1].

1) Applications with constant stack functions allocate stack memory once and then do not change the stack pointer any more. Hence, the amount of stack never changes and the stack function is constant.
2) For periodic stack functions, the stack pointer is increased and then decreased repeatedly, following a fixed pattern. The pattern repeats at a certain frequency. In between those changes, the function may also stay constant for extended periods of time.
3) In rare cases, the stack function is irregular, i.e., no regularity can be trivially identified while further analysis may still be able to identify regularities.

For certain applications, not only one of the aforementioned classes can be observed, but also a change of the class at a certain point in time. The dijkstra benchmark (Figure 2), for instance, first has a constant stack usage, but then switches to an irregular stack usage.

Our ultimate goal is to predict time points at which the stack size is minimal, just by having knowledge about the past stack usage. The analysis provides us with an overview of how stack sizes progress during the execution of common embedded applications. The results in Table I indicate that most applications have stack functions with regularities that can be identified by machine-learning models. When choosing a machine learning model based on this observations, two key requirements should be respected. First, features have to be extracted from the stack usage, reflecting the periodic

[1]Bitcount and Pfor are self-written benchmarks, the other benchmarks are from the MiBench suite.

Constant:	Bitcount (0), Susan (10224), Jpeg (1072), Patricia (240), CRC32 (1072), Blowfish (304)
Periodic:	Basicmath (688), Stringsearch (64), Adapcm (112) Pfor (752), Sha (176), FFT (176)
Irregular:	Dijkstra (688)

Table I. CATEGORIZATION OF 13 EMBEDDED APPLICATIONS[2]

usage patterns also as periodic patterns in the features to enable the model to adopt these patterns. Second, the switching of the stack usage classes should also be detectable by the model. Decision trees [5] seem a good candidate for this purpose, since the stack usage could be determined in the upper levels and specialized sub-trees can adopt the actual pattern subsequently.

V. EFFICIENT WEAR-LEVELING BY PREDETERMINED STACK SIZES

An alternative solution is to analytically determine minimas in the stack usage in advance. By analyzing the stack usage with our analysis framework (Section III) and identifying points during the program execution where the currently used stack is small, the stack relocation can be triggered at these optimal points and the overhead of copying can be reduced. This analysis can be performed in advance and the result can be deployed together with the application.

A. Optimal Stack Usage Determination

To figure out the points during the program execution with the smallest stack, we apply a genetic algorithm that takes the needs for wear-leveling into account. The first important aspect is the target wear-leveling frequency f_{wl}. This value determines the average time distance between two relocations of the stack. For example, $f_{wl} = 5000$ means that in average 5000 instructions should be executed between two relocations of the stack. The second important aspect is the way the determined optimal relocation points are deployed to the wear-leveling algorithms. This is handled by hints, which are placed into the program's source code. A hint is a small set of instructions, which is placed into the original source code and

[2]The applications are categorized based on their stack functions. The numbers in the brackets describe the difference between the minimal and maximal stack size in byte.

triggers the wear-leveling algorithm. Nevertheless, the wear-leveling implementation decides based on the target frequency f_{wl} if a relocation is performed when a hint is given.

The determination of the optimal relocation points is performed by executing a specialized genetic algorithm with post-processing steps. The population of the algorithm is a set of assembly instructions $\mathcal{P} = \{a_0, a_1, ..., a_n\}$, where n denotes the fixed population size. The analysis framework (Section III) is executed for each element of the population at the beginning of each iteration, resulting in a mapping of instructions to instruction ratings $R : \mathcal{P} \rightarrow \mathbb{R}$. In our case, the average stack size ($\frac{\text{cumulative stack size}}{\text{number of calls}}$) is used as a fitness value for each instruction. Instructions which are not called at all result in a null value. Subsequently, a strategy of elitism is applied, and only the k instructions with the best rating are kept in the population $\mathcal{P}$. After this, the current population is checked according to the target frequency f_{wl}. The analysis result for each instruction includes the points during the execution when the instruction was called. This is used to check if multiple instructions are in the population which are called within the same time-frame of the target frequency. In this case, only the best rated instruction of each time-frame is kept. The other instructions are either removed from the population or kept in the population, depending on whether they also are the optimal instruction in another time-frame. At the end of each iteration, a mutation and recombination is performed. Only empty slots in the population are filled with mutated or recombined instructions. Mutations are achieved by adding a bounded random offset to an instruction address from the population. Recombinations are achieved by selecting two random instructions from the population and calculate the arithmetic mean of the instruction addresses. A subsequent sanity check ensures that the calculated address is actually a valid instruction. For the final iteration, the mutation and recombination step is omitted.

B. Deployment of Relocation Hints

After the termination of the genetic algorithm, the output is a set of instruction addresses, which were identified as instructions with an optimal stack size during the execution of the genetic algorithm. These addresses are transformed into the corresponding source code locations by using additional debug information in the compiled binary file. This debug information allows to determine the source file and the line in the source code a specific assembly instruction belongs to. After the source code position is determined for each instruction in the result of the genetic algorithm, the aforementioned hints are placed and the program is deployed back into the original environment. Some special scenarios have to be handled with a special mechanism during this process. For instance, if the genetic algorithm determines the very first instruction of a function as an optimal relocation point, it is pointless to put a hint at the beginning of the function, because the compiler always generates some instructions before the first line of a function. Instead, the hint is placed before each call of the corresponding function in this case.

VI. Evaluation

To evaluate the proposed approaches, we considered two benchmarks. Section VI-A discusses the evaluation of the approximation of stack functions with machine-learning models. If a stack function can be approximated properly, it can also be predicted properly, which allows to determine local minimas in time. Furthermore, we examine our second approach that analyzes the application in advance and deploys the results for the execution in Section VI-B. The evaluation focuses on the saved overhead as well as possible negative effects on the wear-leveling.

A. Model Approximation of Stack Usage

To evaluate the fitting of machine-learning models to the stack function, we used supervised learning algorithms to predict the future stack size in a fixed forward distance. A model was inferred from labeled training data $\{(\vec{x_i}, y)\}$. The value y is a continuous quantity, thus it is a regression problem.

We utilized decision trees (DTs) [5] to predict the future stack size. DTs provide a model to predict values of unseen data, based on the learned information from the training data. A DT is a data structure with nodes, where a node represents a test on a certain attribute. The leaves of the trees provide information about the inferred value.

First, we created features from the application execution, which are used as inputs to the DTs later on. $\vec{I}$ denotes a vector of all instructions that the CPU has already executed. Given the vector $\vec{I} = (i_0, i_1, ...)^T$ of executed instructions, we determined the size of the stack after each executed instruction, using our analysis framework (Section III). This results in an intermediate vector $\vec{S} = (s_0, s_1, ...)^T$ of stack sizes. We subsequently determined the indexes $x_0, x_1, ...$, at which the stack size changes, such that $s_{x_i} = s_{x_i+1} = ... = s_{x_{i+1}-1}$ and $s_{x_i} \neq s_{x_{i+1}}$ for all x_i. We stored $\delta_{SS_i} = s_{x_i} - s_{x_{i-1}}$, which is the amount of change to the stack value when the stack size is modified, and $\#Instr_i = x_i - x_{i-1}$, which is the number of instructions where the stack size did not change, as as a single feature $f_i = (\delta_{SS_i}, \#Instr_i)$. In our analysis, we extracted 128 value pairs from $\vec{I}$ to form a feature vector $\vec{X} = (f_0, f_1, ..., f_{127})^T$, where the pair f_0 is the least recent one, and f_{127} the most recent one. These features represent a part of the stack function in a compressed way.

Four different regression DTs were investigated, which predict the stack size after $S_n = 250, 500, 750, 1000$ instructions. During runtime, the predicted stack sizes of the model can be monitored. Whenever the predicted, future stack size is smaller than the current stack size, relocations should be delayed further, if possible. Once the predicted stack size increases, the relocation should be triggered after additional S_n instructions.

The datasets to train the models were generated while executing the application in our analysis framework (Section III), by sampling at random time points to determine the future stack size relative to those random time points. In our experiments, the sampling was realized as follows: Before the application execution, a random number $N_r \in \{0, ..., \text{maximum value}\}$ is generated. We used

maximum value=1000 in our evaluation. The application was executed until the first feature vector $\vec{X}$ was filled with 128 elements. Additionally, the stack size after S_n additional instructions, denoted as y, was recorded and $(\vec{X}, y)$ was stored as one training sample. During the subsequent N_1 instructions, the feature vector $\vec{X}$ was updated by discarding old and adding new feature pairs. Thus, the length of the feature vector was always 128 elements. When N_1 instructions were executed, the stack size S_n instructions later was stored together with the current $\vec{X}$ as a training sample. The process was repeated with a new random number N_2[3]. This procedure continued until a certain number of instructions was reached. In our experiments, we recorded several thousand training samples of the application execution. The test samples were recorded from disjoint execution intervals. This allowed us to train a model with samples, recorded by executing the application for a limited time, and we can apply the model for indefinite time when deploying the application in a production setting.

Afterwards, we loaded the recorded samples into sklearn [11], and trained and tested a decision tree for every application scenario. The trees had a depth no greater than eight. The dataset consist of thousands of samples, e.g., to train a DT that predicts the stack size after 250 instructions for the Pfor application, and we used 5797 training samples and 3803 test samples.

We estimated the precision of the trained models using the mean absolute error (MAE), the mean squared error (MSE), and the coefficient of determination (R^2). The results for two applications are presented in Table II, .

For the Pfor application, the measures indicate high prediction accuracy for all S_n values shown, since the MAE is at most 1.12 while the maximum stack size is 848. Hence, the error ranges at 0.13% of the stack size.

For Dijkstra, the MAE is approximated one order of magnitude larger compared to Pfor (MAE is at most 11.94, the maximum stack size is 160, thus the error ranges at 7.46%).

In conclusion, due to the periodicity of many embedded applications, simple models are able to produce accurate predictions. In the case of highly irregular stack usage, our models are unable to provide an accurate prediction. However, we believe that machine learning models in general will suffer from predicting data without identifiable patterns.

Alternatively, we trained smaller fully connected neural networks (FCNNs), i.e., multi-layer perceptrons with a few layers and less than 100 neurons. The results indicate errors that are larger by an order of magnitude compared to the DT results.

We limited our evaluation to two example applications (Pfor and Dijkstra). These applications represent two classes of stack functions, i.e., periodic and irregular stack usage. Performing the evaluation for constant stack functions is a trivial setup for all our tested models and, hence, omitted in the evaluation.

[3]To only execute the application once and omitting the need to roll back in the instruction stream, the implementation used a set of overlapping timers, which count N_r and S_n

	R^2	MAE	MSE
Pfor			
250	0.98	0.91	207.20
500	0.99	0.39	38.74
750	0.99	0.70	65.05
1000	0.96	1.12	337.3
Dijsktra			
250	0.45	5.97	276.6
500	−0.27	11.84	639.60
750	−0.28	11.94	645.7
1000	−0.66	7.42	284.80

Table II. THE RECORDED ACCURACY MEASURES FOR THE TWO APPLICATIONS PFOR AND DIJKSTRA, EACH PREDICTING THE STACK SIZE IN 250,500,750 AND 1000 INSTRUCTIONS

B. A Priori Application Analysis

To evaluate our approach that combines a genetic algorithm with the analysis framework to determine optimal time points to perform a stack relocation (i.e. a less usage of the stack), we set up a benchmark application and performed an end-to-end evaluation. In detail, the application was separated from the original execution environment and analyzed by the genetic algorithm. The resulting hints were integrated into the source code, the application was deployed back into the original environment, and executed again. As the original execution environment we chose a simulator (gem5 [3]), which allowed us to measure the total amount of memory accesses. Hence, we evaluated the total amount of memory write accesses for the wear-leveling algorithm without hints and compared them to the same setup with the hints, provided by the genetic algorithm. This setup also includes several overheads, caused by the additional source code for the hinting itself.

We used a data decompression benchmark application, which processes lightweight compressed data (PFOR compression [14]) in a stream-like manner and decompresses and aggregates them into a local data structure. The genetic algorithm was configured with a population size of $n = 200$ elements, and the elitism strategy kept the best $k = 100$ elements in each iteration. The target frequency for stack relocations was configure to $f_{wl} = 10000$ and the boundary of the random mutation offset was 50 instructions into each direction. The final results of the genetic algorithm were inserted into the application source as hints for the wear-leveling system. A hint is a simple call to the responsible wear-leveling function in the runtime environment. Figure 3 shows the end-to-end result of the wear-leveling, when the genetic algorithm is used to determine the best relocation points. We observed that the placing of the hints had no negative effect on the wear-leveling itself, but that the write count is dropped down overall the memory on the blue plot (with hints), compared to the green plot (without hints). More detailed, the wear-leveling caused a total write overhead of 9.96% without any optimization. Applying the method that is presented in this paper reduced the overhead to 1.41%, which is a saving of 85.83% on the

[1]The figure shows the memory write count distribution of wear-leveling with and without hints. The x axis displays the memory space, the y axis the total write count to the memory.

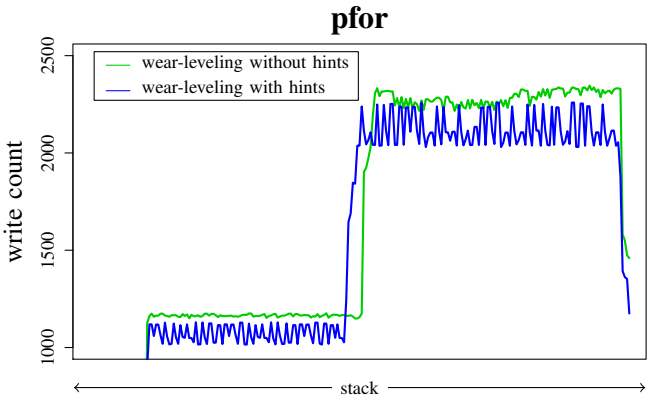

Fig. 3. Memory write access distribution[1]

total write-overhead. As mentioned before, the wear-leveling quality is not affected by our method and the optimization is achieved by carefully choosing the points in time to perform a stack relocation.

VII. Future Outlook

The necessity to analyze an application a priori to perform efficient wear-leveling (Section V) is a significant drawback and may not be possible in every scenario. Furthermore, our results in Section VI-A show that simple machine-learning models are capable of estimating the stack function of most applications accurate enough to deliver a meaningful prediction. This motivates the design of an adaptive online controller, which aids the process of wear-leveling.

Realized as a hardware controller close to the CPU, the CPU internal state (e.g. the stack pointer) is easy to acquire. This can be used to train a simple machine-learning model (e.g. a decision tree) during the application execution. Again, sampling can be used instead of training the model with every executed instruction to relax the timing requirements for the controller. Once the model inside of the controller is trained with a reasonable amount of data, it can predict the future stack size. Now, a wear-leveling algorithm may interact with the controller to determine the most efficient points to perform stack relocation. The wear-leveling algorithm could also be integrated further into the controller. The controller can keep track of the future stack size automatically and only interrupt the CPU whenever a good point for a stack relocation is reached. The interrupt handler has to decide subsequently if a wear-leveling action really should be performed during the interrupt handling.

A further hybrid solution to omit the initialization phase of the controller is to ship the application with a parameter set of a pre-trained model. Thus, the model can be used immediately to predict future stack sizes. Still, the model can be trained from time to time on the execution data to react to a changed behavior of the application and to make the prediction more precise.

We plan to prototype the hardware controller on a CPU / FPGA combination to evaluate the online trainability of the previously evaluated machine-learning models.

Acknowledgement

This paper has been supported by Deutsche Forschungsgemeinshaft (DFG), as part of Project OneMemory (Project number 405422836), and the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A1 (http://sfb876.tu-dortmund.de).

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] *ptrace(2) Linux Programmer's Manual*, October 2019.

[3] 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.

[4] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging nvm: A survey on architectural integration and research challenges. *ACM Trans. Des. Autom. Electron. Syst.*, 23(2):14:1–14:32, November 2017.

[5] Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olshen. *Classification and regression trees*. Routledge, 1984.

[6] Sangyeun Cho and Hyunjin Lee. Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 347–357, New York, NY, USA, 2009. ACM.

[7] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

[8] 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.

[9] 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.

[10] 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.

[11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *J Mach Learn Res*, 12:2825–2830, 2011.

[12] 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.

[13] 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.

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