Can Wear-Aware Memory Allocation be Intelligent?

Christian Hakert, Kuan-Hsun Chen, Jian-Jia Chen
Department of Computer Science, TU Dortmund, Germany
https://ls12-www.cs.tu-dortmund.de/

Citation: https://doi.org/10.1145/3380446.3430624

Can Wear-Aware Memory Allocation be Intelligent?

Christian Hakert, Kuan-Hsun Chen, Jian-Jia Chen
christian.hakert@tu-dortmund.de,kuan-hsun.chen@tu-dortmund.de,jian-jia.chen@cs.tu-dortmund.de
TU Dortmund, Design Automation for Embedded Systems Group

ABSTRACT

Many non-volatile memories (NVM) suffer from a severe reduced cell endurance and therefore require wear-leveling. Heap memory, as one segment, which potentially is mapped to a NVM, faces a strong application dependent characteristic regarding the amount of memory accesses and allocations. A simple deterministic strategy for wear leveling of the heap may suffer when the available action space becomes too large. Therefore, we investigate the employment of a reinforcement learning agent as a substitute for such a strategy in this paper. The agent's objective is to learn a strategy, which is optimal with respect to the total memory wear out. We conclude this work with an evaluation, where we compare the deterministic strategy with the proposed agent. We report that our proposed agent outperforms the simple deterministic strategy in several cases. However, we also report further optimization potential in the agent design and deployment.

CCS CONCEPTS

• **Computing methodologies** → **Markov decision processes**;
• **Hardware** → *Aging of circuits and systems*;

ACM Reference Format:
Christian Hakert, Kuan-Hsun Chen, Jian-Jia Chen. 2020. Can Wear-Aware Memory Allocation be Intelligent?. In *2020 ACM/IEEE Workshop on Machine Learning for CAD (MLCAD '20), November 16–20, 2020, Virtual Event, Iceland.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3380446.3430624

1 INTRODUCTION

Arising technologies for *non-volatile main memory* (NVMM), such as phase change memory (PCM) or ferroelectric RAM (FeRAM), require additional maintenance and wear-leveling due to the severe reduced *cell write endurance* compared to classic DRAM or SRAM. The literature covers by now a variety of wear-leveling approaches, which operate on various levels, for instance inside the memory controller, in the operating system or even on the application level. *Heap memory*, however, depends strongly on the usage of the application, i.e. how often the application performs `malloc` calls and how the application uses the allocated memory. Many so-called wear-aware allocators are proposed in the literature, which aim to place memory requests not only along the objective of fast allocation time or low fragmentation, but also with the target objective to optimize the total wear out of the underlying physical memory. A straightforward approach is to implement a deterministic strategy, where for instance a memory request is always placed to the coldest, i.e. the least often written, memory region. In addition, not only the allocation process can be wear-aware, but also after answering the malloc requests, physical memory locations may be exchanged by utilizing the *memory management unit* and *virtual memory*. This leads to runtime decisions, which also have to encounter the wear out of the memory.

In this paper, we propose a novel concept for runtime remapping and investigate a heap allocation design where not only the physical location of entire *memory pages* can be remapped, but also allocated memory portions (*memory chunks*), which may be smaller than one page, can be recombined into different memory pages. This yields an even larger degree of flexibility to map memory chunks to the physical memory. Given this degree of flexibility, making the right decision of mapping chunks to a page and to a physical page respectively becomes an increasing complex task and one interesting research question is whether a simple strategy makes good decisions or whether a self learning, decision making *machine learning agent* can even further improve the wear-leveling.

To answer this question, we propose one possible design of such a machine learning agent. The design is realized as a *reinforcement learning* agent, since it is supposed to learn a good strategy for runtime remapping on its own. For training, the agent only needs to be provided with the immediate reward from the system, which can be generated from the current memory wear out. More specifically, our agent is design with *Q Learning* [15] which is a popular approach for *model free* learning and fits our needs. We aim to find a minimal state description for the agent, which still fulfills the *Markov property* but does not carry much overhead. This approach explicitly targets small embedded systems, since usually a small and fixed set of software is executed and not many programs are executed in an interleaved manner. Our design intends to learn the memory access properties of the software set and optimizes among them. We evaluate our proposed agent on a set of benchmarks from the Olden benchmark suite [1, 12], which are dedicated benchmarks for dynamic data structures and therefore for heap memory. We compare the runtime decisions of our proposed agent with a possible deterministic design for runtime remapping, which always remaps the most heavily written memory chunk to the next fitting free slot, in the evaluation of this paper.

Our novel contributions:

- The investigation of a novel heap memory allocation concept, where allocated memory chunks can be recombined to memory pages after the allocation
- A design proposal for a reinforcement learning agent to decide for runtime recombination of memory chunks, which follows the allover target of optimizing the total memory wear out.
- An evaluation of our proposed agent, which compares the agents decisions with a deterministic recombination strategy.

MLCAD '20, November 16–20, 2020, Virtual Event, Iceland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7519-1/20/11...$15.00
https://doi.org/10.1145/3380446.3430624

2 RELATED WORK

Wear aware heap allocation is tackled with several approaches in the literature. Wang et al. propose NVMalloc, which provides a memory allocation interface [14]. The strategy is to not allocate the same physical memory two times within one time interval. This relaxes the stress on the memory. Yu et al. push this even further with WAlloc [18]. The allocation of memory requests follows the *Less Allocated First Out* policy, which means that for each and every memory request the less allocated physical memory block is chosen to serve the request. This levels the number of allocations across all memory blocks. Huang et al. include write count approximations in Quail [6], which also provides a memory allocation interface. Quail monitors the number of write accesses to virtual memory pages and remaps the page to a free physical page once it exceeds a certain amount of write accesses. UWLalloc further improves the management of physical memory and the detection of hot memory blocks [9]. These blocks are excluded from the allocation at a certain time. Although all these allocation algorithms can grow rather complex and are able to achieve several improvements in the wear leveling, they still make decisions by a deterministic policy. To the best of our knowledge, no wear-aware allocator exists which tries to replace this design principle with a more complex machine learning policy. Deng et al. propose Memory Cocktail Therapy (MCT) [2], which utilizes various machine learning approaches to choose a proper combination and configuration of existing NVM maintenance mechanisms. This approach, however, does not aim to find optimized policies for wear-leveling itself, it rather selects and configures pre-defined wear-leveling algorithms to achieve the desired objective.

To design a memory allocator, which uses machine learning to make advanced wear-aware decisions, several machine learning algorithms could be considered. Supervised learning exists in many flavours but always follows the concept of learning correlations from given training examples [8]. In order to design a wear-aware memory allocator with supervised learning, the machine learning algorithm would need to be trained with training data from an ideal wear-aware memory allocator. Therefore we do not consider supervised learning here. Unsupervised learning [5] in contrast would not require training samples from a perfect allocator and still could classify the memory allocation requests along criteria which correspond to the memory wear out. Nevertheless the mapping decision can be based on these criteria, it still has to follow some given strategy. Reinforcement learning [13], as a remaining algorithm, interacts with the real system during training and requires a reward, which can be generated from the environment, without following some given wear-leveling policy. The reinforcement learning agent then aims to find an optimal behavior policy on its own to achieve the best reward. Therefore we use reinforcement learning in this work to learn a strategy to place memory requests to physical locations, based only on a simple reward mechanism.

3 Q LEARNING

This section gives a broad overview about the reinforcement learning methods, on which the proposed agent in this paper is based on. Q learning, as one flavour of reinforcement learning, is introduced by Christopher Watkins in [16] and is shown to converge under given conditions in [15]. Q learning is a model-free reinforcement learning approach, therefore no model of the environment is required. An agent learns a policy for optimal behavior based on immediate rewards, which are provided by the environment.

3.1 Environment Representation

Since Q learning is model-free and does not require a model of the environment to be provided, it assumes a generic structure of the environment, which can be described by a Markov decision process (MDP). The environment is assumed to reside in a given state $s_n \in S$ at the time point n. From each state, the agent decides for an action $a_n \in A$, which will push the environment into a successor state $f \in S$. This transition must only depend on the current state s_n and the chosen action a_n:

$$Prob_a(s, f) = Prob(s_{n+1} = f \mid s_n = s, a_n = a) \tag{1}$$

Furthermore, the environment is assumed to provide an immediate reward r_n at each time point n, which again only depends on the current state s_n and the chosen action a_n. For practical application, the problem, which the agent should solve, has to be observable up to a certain degree, such that a state representation can be generated, which follows the aforementioned condition. Additionally, an immediate reward has to be generated by the training environment, which follows the mentioned condition.

Once states and actions are know, the central source of information for the agent is to know the *total expected reward* $Q : S \times A \longrightarrow \mathbb{R}$, which does not only include the immediate reward r_n, earned for choosing a_n in state s_n, but also considers the agents future decisions and the immediate rewards for them. As the future decisions and thus the future rewards depend on the policy π along which the agent chooses decisions, Q is also dependent on π. Throughout this paper, the proposed agent follows the ϵ-greedy policy [17], i.e. it always tries to select the action a_n, where the total expected reward is maximal:

$$a_n = \underset{a \in A}{\operatorname{argmax}}\, Q(s_n, a) \tag{2}$$

Randomly, with a probability of $\frac{\epsilon}{|A|}$, where ϵ is a given parameter, a random action is chosen from time to time, which ensures exploration. This leads to the specific definition of the Q function:

$$Q(s_n, a_n) = r_n + \gamma \cdot \underset{a \in A}{max}\, Q(s_{n+1}, a) \tag{3}$$

γ is the total expected reward *discount factor*, which prevents the Q function from considering an infinite time horizon when $0 \leq \gamma < 1$. Assuming the Q function is perfectly known, the agent decides for the best decision in every time step. Thus, the allover goal of Q learning is to learn a proper surrogate function Q' based only on the given immediate rewards, which is then used to support the agents decisions.

3.2 Q Function Approximation

During runtime, the agent is initialized with some surrogate function Q'. The agent then defers actions based on this function and therefore receives immediate rewards from the environment. This forms one *training tuple* for every executed action:

$$(s_n, a_n, r_n, s_{n+1}) \tag{4}$$

The tuple contains the current state, the taken action, the earned immediate reward and the successor state. Each of these tuples can be used to compute an update of the surrogate function:

$$Q'_{update}(s_n, a_n) = r_n + \gamma \cdot \max_{a \in A} Q'(s_{n+1}, a) \quad (5)$$

This updated value then can be used to incrementally train the surrogate function:

$$Q'(s_n, a_n) \leftarrow Q'(s_n, a_n) + \lambda \cdot (Q'_{update}(s_n, a_n) - Q'(s_n, a_n)) \quad (6)$$

λ here denotes the *training rate* for the surrogate function.

Further modifications exist to improve the quality of the agent further. One is to employ an additional surrogate function Q'' as the *target function* [11], which decouples the training of the *prediction function* Q' from the values of the prediction function itself. The updates for the prediction function then are calculated in a slightly different manner:

$$Q'_{update}(s_n, a_n) = r_n + \gamma \cdot \max_{a \in A} Q''(s_n + 1, a) \quad (7)$$

From time to time with a given frequency, the target function is replaced with a recent snapshot of the prediction function. Another modification, which is also employed in the later proposed agent, is *experience replay* [10]. Instead of computing an update value for the prediction function, whenever a training tuple comes in, training tuples are first stored in a *replay buffer*. To compute update values, a random sample of training tuples is extracted from the replay buffer then and the update values are computed for these training tuples. This can help to overcome temporal correlation in the training tuples.

The final question is how the agent encodes and stores the surrogate function Q' and Q''. An intuitive solution would be to maintain a value table $S \times A \times \mathbb{R}$, which holds one row for each value combination of the function. With a huge state and action space this may lead to serious efficiency problems and furthermore, updates only apply to a single row of the table. An alternative is to use regression models, which try to fit some function to the computed update values of Q'. In this paper, we use multi layer perceptron (MLP) neural networks for this approximation. Instead of letting the network learn the function $Q' : S \times A \to \mathbb{R}$ based on the update values, we restructure the network to predict all total expected rewards for all actions at once:

$$Q' : S \to \mathbb{R}^{|A|} \quad (8)$$

The training data for this network then is inferred by modifying only the corresponding output element with the updated Q' function value. The usage of the MLP regression model eventually introduces an additional tuning parameter, i.e. the learning rate of the neural network. Updating the target function, however can be done by simply copying the weights from the prediction function to the target function from time to time.

4 WEAR-AWARE ALLOCATION AGENT

In order to adapt Q learning, two basic requirements are as follows: 1) the state of the problem needs to be described in a description, which fulfills the Markov property (Equation (1)), and 2) each combination of state and chosen action must lead to a particular immediate reward. Wear-leveling, which is performed for the allocation of heap memory, resides as a software application in the scope of the allocation mechanism of the operating system. Considering the entire operating system memory footprint may provide a sufficient

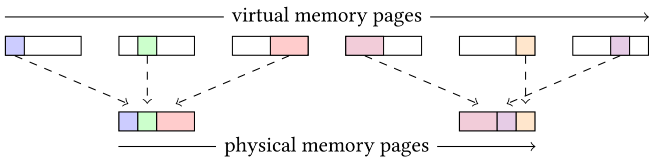


Figure 1: Virtual memory page overlaying

state description, which fulfills the Markov property but is practically impossible due to the high complexity. Therefore, it is crucial to design a state description which fulfills the Markov property but carries no unused overhead. We detail our design and state description for heap wear-leveling in this section.

4.1 Heap Allocation Structure

When memory requests are arbitrarily placed in the memory space, remapping to other physical addresses is usually limited to the granularity of the virtual memory subsystem. In order to provide an increased freedom of remapping memory chunks to physical locations, we propose a novel heap allocator structure, which yields a largely increased action space for relocating memory chunks to new physical locations.

We allocate one or more separate *virtual memory pages* for each allocation request which are subsequently mapped to the allocated *physical memory*. By aligning the allocated *memory chunks* at fixed boundaries, we can map multiple virtual memory pages to the same physical memory page, as shown in Figure 1. This also allows a remapping during runtime, as long as no overlapping happens in the physical memory pages. Since this mechanism is implemented with virtual to physical address translation, it is transparent to the application and does not require any additional or special application support.

The allocator splits the memory pages into equal sized *subpages* with the size of $4096 \cdot \frac{1}{2^n}$, such that chunks with different sizes can be still combined into one physical memory page. The free space management within pages then is done with the buddy allocation strategy [7]. Please note that the initial assignment of a newly requested memory chunk is not decided by the agent but is done randomly.

4.2 State Representation

Since the state representation has to cover the deterministic state of the allocator status to fulfill the Markov property, it has to be clarified on which information wear-leveling and relocation decisions are based on. The first important aspect is the absolute age of physical memory pages, which should be considered for wear-leveling decisions. The second aspect is the relative age of memory chunks, i.e. how intensive a memory chunk aged the memory since the last remapping. This encodes the write behavior of the application. Since this can change during runtime, it has to be included in the state. Finally, the current mapping from virtual memory chunks to physical pages should be included in the state representation, such that chunks with a high relative age can be mapped to physical pages with a low absolute age, for instance.

To summarize all these information into a compact state description, which can be easily parsed by the MLP regressor, we construct a state vector to describe the state as shown in Figure 2. The vector holds 2 consecutive entries for every subpage. If, for instance, 10 pages exist in the system and each page has 8 subpages, the

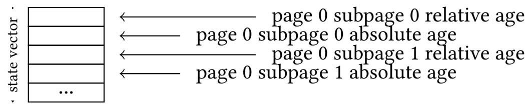

state vector
...
page 0 subpage 0 relative age
page 0 subpage 0 absolute age
page 0 subpage 1 relative age
page 0 subpage 1 absolute age

Figure 2: State vector

vector holds 160 entries. Entries at even indices hold the relative age of the chunk, which is mapped to the physical subpage at the index i = vector index/2. Entries at odd indices hold the absolute age of the physical subpage at index i. For all entries, the values are normalized according to the maximum relative memory chunk age, respectively the maximum absolute subpage age. Note that we gather the age information by runtime sampling of write accesses, as proposed by Hakert et al. [3]. Therefore, the realization is mostly independent from specific hardware features.

4.3 Agent Actions

The state vector is used as the input for the agent, i.e. as the input for the MLP regressor. The output then is a vector of total expected rewards for each possible action. Thus, the number of actions needs to have a fixed size, such that the regressor architecture does not need to change during runtime. We implement the actions in such a way, that one action is available per possibly existing memory chunk. The number of possibly existing memory chunks is a fixed number, since there cannot be more memory chunks than subpages. Each action then leads to the following steps:

(1) Identify the mapped memory chunk targeted by this action (the chunk may cover multiple subpages or may not exist).
(2) Search for other free subpages to map this chunk to (next fit).
(3) Apply the remapping of the chunk to the new target.

4.4 Agent Realization

Plugging all together, the wear-aware remapping agent for heap allocated memory is implemented straight forward. The application is interrupted after a certain amount of memory write accesses and the state vector is generated from the access count approximations. From the inferred total expected rewards, the desired action is chosen according to the ϵ-greedy policy. The action is applied and the application continues to run until the next *maintenance interrupt*. Subsequently, the allover wear-out from the memory is determined and an average indicator is calculated on the count of writes per byte according to Equation (9).

$$r = \frac{\text{mean\_write\_count}}{\text{max\_write\_count}} \tag{9}$$

This indicator builds the immediate reward for the action chosen in the last maintenance interrupt. When the memory is not properly wear leveled, this number is near to 0, when write accesses are distributed mostly uniform, this number is near to 1. If the agent selects an action, which targets a not existing chunk, the immediate reward is set to -1. The training tuple is formed and stored in the replay buffer. With a configured rate, the agent then samples tuples from this buffer and updates the model.

5 EVALUATION

In order to evaluate our proposed reinforcement learning agent we compare our proposed wear-aware agent (Section 4) with a simple deterministic strategy, but also with a blind random strategy. As benchmark applications, we chose three applications from the Olden benchmark suite [1, 12]: The "Barnes & Hut N-body force computation" (**BH**), the "Electromagnetic wave propagation in a 3D object" (**EM3D**) and the "Perimeters of regions in images" (**PERIMETER**). We selected these benchmark applications because they repeatedly perform allocations and de-allocations. We limit the evaluation to 3 benchmarks due to the effort for instrumenting the code and the computational effort for the simulations.

$$AE = \frac{\text{mean\_write\_count}}{\text{max\_write\_count}} \tag{10}$$

The evaluation compares the strategies along their achieved endurance Equation (10) over simulation time. This number indicates the current wear leveling of the memory. This metric differs from Equation (9) in the fact that it is computed on the real write count per memory byte and not based on the runtime estimation.

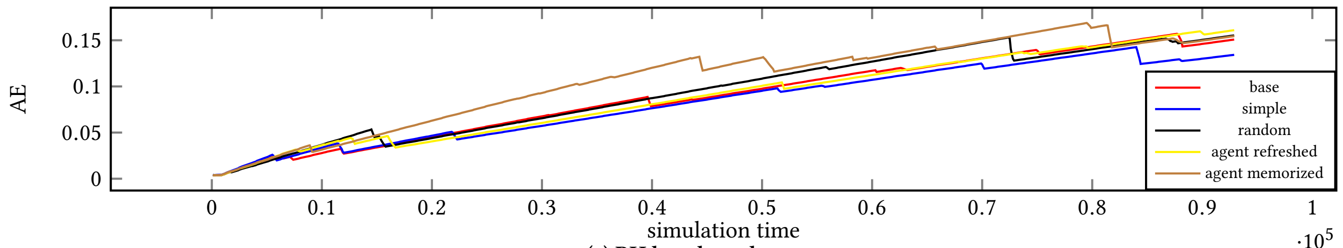
5.1 Simulation Setup

We execute the benchmarks in a realistic memory simulation setup [4] to get the full memory access trace as a result. We further instrument the *malloc* calls in such a way, that we get the memory requests as an additional simulation result through a debug channel. This allows us subsequently to form a set of chunks, each benchmark allocates and a series of memory accesses, targeting each chunk. We carry this information into an in-house simulation of our memory allocator, where we virtually place memory chunks at according physical locations while encountering all technical limitations (subpages always have a fixed offset within pages). The execution of the benchmark is simulated by redirecting the recorded memory accesses according to the current mapping. We invoke the maintenance strategy with a fixed frequency (i.e. after every 1024th write access) and modify the mapping accordingly. The state representation for the agent and the immediate rewards are generated as described in Section 4.

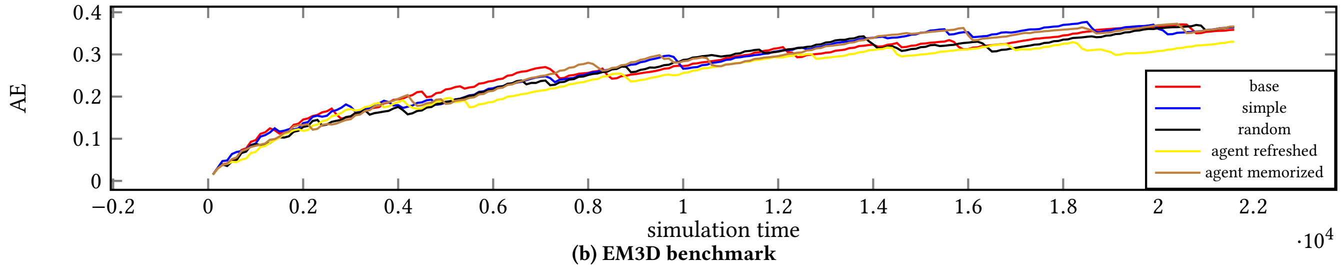
Throughout intensive experiments, we found a set of parameters for the reinforcement learning agent, which is used in all simulations. The MLP regressor has the entire state as the input layer, which is #pages $\cdot$ 8 $\cdot$ 2 since there are 8 subpages per page. The output layer has the size of #pages $\cdot$ 8 since there is one possible action per chunk, respectively per subpage. The MLP has two hidden layers, the first with the same size as the input layer and the second with the same size as the output layer. The replay buffer contains up to 500 elements, where 4 tuples are extracted in each training step, which happens every 8th inference of the agent, i.e. after 8 training tuples are appended. The target function is updated after 16 training intervals; The learning rate λ is set to 1 initially and slowly decreases to 0.1 during training; The total expected reward discount factor γ is set to 0.8. Eventually, the MLP training rate is set to 0.2.

The memory space is $64 \times 4kB$ for the BH and EM3D benchmark and $256 \times 4kB$ for the PERIMETER benchmark. For all strategies, memory chunks are randomly assigned to subpages on their initial allocation. For every benchmark application, we run the following maintenance strategies:

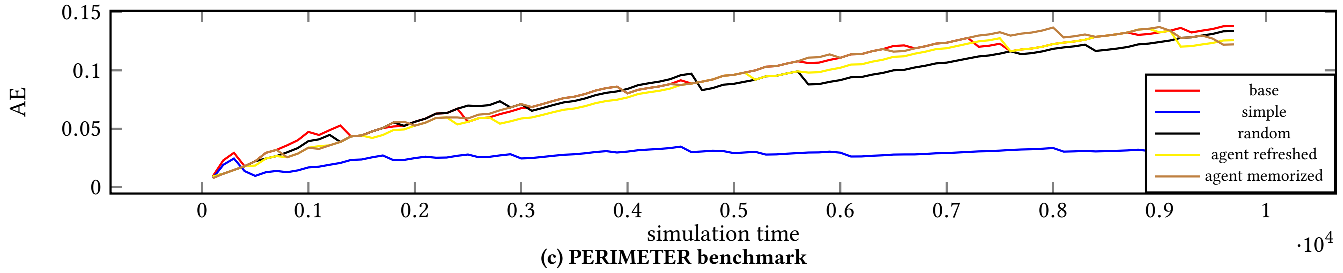
(1) **base**: No runtime maintenance is applied.
(2) **simple**: The "hottest" chunk is determined by finding the subpage with the highest absolute age. The according chunk action



(a) BH benchmark



(b) EM3D benchmark



(c) PERIMETER benchmark

Figure 3: Achieved endurance for memory simulations

is applied, i.e. the chunk is relocated to the next fitting free subpage.

(3) **random**: A uniform sampled random chunk is chosen and the chunk action is applied.

(4) **agent refreshed / memorized**: The reinforcement learning agent selects the chunk action. Agent refreshed means that the agent starts with an empty weight file while agent memorized means that the agent already was executed on the benchmark once and thus is pre-trained.

5.2 Result and Discussion

Figure 3 illustrates the *achieved endurance (AE)* (Equation (10)) during the simulation time for all benchmarks and maintenance strategies. Note that the simulation time for the benchmarks is different, which stems from the varying execution time of the algorithms in the benchmarks. Table 1 summarizes the improvement of the mean values of the achieved endurance for each a pair of two strategies. First, it can be observerd that the general quality of the wear-leveling is very benchmark dependent. For the PERIMETER benchmark huge differences in the maintenance strategies can be observed, while for EM3D all strategies achieve somewhat similar results. The simple strategy turns out to not improve, but rather degrade the achieved endurance for PERIMETER. The simple strategy of always targeting the "hottest" chunk may suffer when the access behavior is changing a lot and the "hottest" chunk changes regularly. In this case, accesses are already distributed without any maintenance. For EM3D, in contrast, the simple strategy does not degrade the endurance and improves by 0.4% in average. For BH the simple strategy slightly degrades the endurance by 0.6% in average. Due to the fact that for all strategies memory chunks are randomly allocated to physical locations on their initial allocation, already a good, randomized wear-leveling happens even without any runtime maintenance strategy. For the investigated benchmarks, it can be clearly reported that the simple design cannot achieve a significant endurance, indeed the endurance is degraded.

Considering the achieved endurance of the memorized agent, an improvement can be observed for most benchmarks. For BH, the agent achieves 1.7% more endurance than the baseline in average, for EM3D 0.2% and for PERIMETER 0.03% less endurance. In comparison to the simple strategy, BH improves the endurance by 2.4% in average, PERIMETER by 6.2% and EM3D degrades the endurance by 0.2% in average. It can be further observed that the second execution of the agent improves upon the first execution of the agent. This draws the conclusion that a crucial key to the performance of the agent is the amount of available training data. This conclusion is also supported by investigating the *mean squared error (MSE)* of the training process of the MLP. For EM3D, the MSE has an average value of 85173 during the first run and a mean of 2698 during the second run. Also for PERIMETER. the mean MSE is 2159173 during the first run and 1295676 during the second run. Only for BH, the mean MSE value is bigger for the first run (3173) then for the second run (23364). First, the improvement of the MSE in the second run indicates that the agent already reacted better to some scenarios then in the first run, therefore also the allover achieved endurance improves. Second, the absolute high values of

	BH	EM3D	PERIMETER
base → simple	−0.659%	0.413%	−6.292%
base → agent ref.	−0.162%	−2.256%	−0.514%
base → agent mem.	1.750%	0.202%	−0.028%
base → random	0.457%	−0.585%	−0.576%
simple → agent ref.	0.643%	−2.669%	5.778%
simple → agent mem.	2.408%	−0.211%	6.263%
simple → random	1.115%	−0.998%	5.716%
random → agent ref.	−0.473%	−1.671%	0.062%
random → agent mem.	1.293%	0.787%	0.547%

Table 1: Mean AE improvements[1]

the MSE indicate that the agent by far does not behave optimal and there is improvement potential.

5.3 Discussion

Summarizing the observations from the conducted experiments, we see that the simple wear-leveling strategy does not perform optimally in our benchmarks. Even deciding randomly for relocations achieves a better result, which highlights the random strategy as a universal applicable strategy. Since the action space is limited and for every performed wear-leveling action the memory mapping changes, random intuitively seems quiet suitable. This is because over time all memory chunks and all subpages are targeted. However, although the proposed reinforcement learning agent cannot outperform the random strategy by far, it adapts to the specifics of the benchmarks. The memorized agent performs better than the random strategy in all benchmarks.

The results also point out that training data is an essential key to the performance of the reinforcement learning agent. The agent starts to make decisions completely untrained and only gets the runtime behavior of the memory as an input. During all the benchmark run, this leads to a few thousands of training tuples only, which are not sufficient to fit the internal MLP model properly to the estimated Q function. With a pre-trained agent both, an improvement of the wear-leveling and a better fitting of the internal model can be observed. For further improvement of the agent, a better training scenario should be considered. The agent could be pre-trained offline or the random strategy could be applied first until the internal model of the agent approximates the Q function up to a certain degree. Although we do not have these methods in our proposed agent, we still report that the agent properly adapts to the situation and achieves reasonable wear-leveling.

6 CONCLUSION

In this paper, we investigate wear-aware heap memory allocation for arising non-volatile main memories. We consider a novel heap allocation structure which yields a larger action space for the combination and recombination of single allocated memory portions to physical memory locations. Due to the growing action space, we challenge simple deterministic designs of wear-aware heap allocation and propose a possible design of an intelligent heap allocation, which is realized with a reinforcement learning agent. We choose reinforcement learning because we only need to extract the immediate reward as training data from the environment and do not need to generate supervised training tuples.

Our evaluation points out that the simple design may degrade the memory lifetime, while even a uniform random strategy can achieve a reasonably good result. Our reinforcement learning agent also adapts to the specific memory access patterns of the benchmark application and outperforms the random strategy.

[1]Each row of the table compares two strategies. The difference of the achieved endurance of the strategies mentioned in the firsat column is averaged over the simulation time. Agent mem. here always indicates the memorized agent. Agent ref. indicates the refreshed agent.

7 OUTLOOK

In this work, we observe a significant improvement in the allocator when it already was trained on the benchmark once before. Towards this, one approach would be to pre-train the agent on recorded data. Another approach would be to execute the random strategy at first and train the agent meanwhile without letting it make strategy decisions. Furthermore, it has to be investigated if the agent only improves when pre-trained on the given benchmark application or if it even can improve when pre-trained on other application.

ACKNOWLEDGEMENT

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

REFERENCES

[1] M. C. Carlisle and A. Rogers. Software caching and computation migration in olden. Technical Report TR-483-95, Princeton University.

[2] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T Chong. Memory cocktail therapy: a general learning-based framework to optimize dynamic tradeoffs in nvms. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 232–244, 2017.

[3] Christian Hakert, Kuan-Hsun Chen, Pual R Genssler, Georg von der Brüggen, Lars Bauer, Hussam Amrouch, Jian-Jia Chen, and Jörg Henkel. Softwear: Software-only in-memory wear-leveling for non-volatile main memory. *arXiv preprint arXiv:2004.03244*, 2020.

[4] Christian Hakert, Kuan-Hsun Chen, Mikail Yayla, Georg von der Brüggen, Sebastian Blömeke, and Jian-Jia Chen. Software-based memory analysis environments for in-memory wear-leveling. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 651–658. IEEE, 2020.

[5] Geoffrey E Hinton, Terrence Joseph Sejnowski, Tomaso A Poggio, et al. *Unsupervised learning: foundations of neural computation*. MIT press, 1999.

[6] Kaixin Huang, Yijie Mei, and Linpeng Huang. Quail: Using nvm write monitor to enable transparent wear-leveling. *Journal of Systems Architecture*, 2020.

[7] Kenneth C Knowlton. A fast storage allocator. *Communications of the ACM*, 1965.

[8] Erik G Learned-Miller. Introduction to supervised learning. *I: Department of Computer Science, University of Massachusetts*, 2014.

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

[10] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.

[12] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2), March 1995.

[13] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction.

[14] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*.

[15] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*.

[16] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards.

[17] Michael Wunder, Michael L Littman, and Monica Babes. Classes of multiagent q-learning dynamics with epsilon-greedy exploration. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. Citeseer, 2010.

[18] S. Yu, N. Xiao, M. Deng, Y. Xing, F. Liu, Z. Cai, and W. Chen. Walloc: An efficient wear-aware allocator for non-volatile main memory. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*.