Memory Cocktail Therapy: A General Learning-Based Framework to Optimize Dynamic Tradeoffs in NVMs

Zhaoxia Deng
UC Santa Barbara
zhaoxia@cs.ucsb.edu

Lunkai Zhang
University of Chicago
lunkai@uchicago.edu

Nikita Mishra
University of Chicago
nmishra@cs.uchicago.edu

Henry Hoffmann
University of Chicago
hankhoffmann@cs.uchicago.edu

Frederic T. Chong
University of Chicago
chong@cs.uchicago.edu

ABSTRACT

Non-volatile memories (NVMs) have attracted significant interest recently due to their high-density, low static power, and persistence. There are, however, several challenges associated with building practical systems from NVMs, including limited write endurance and long latencies. Researchers have proposed a variety of architectural techniques which can achieve different tradeoffs between lifetime, performance and energy efficiency; however, no individual technique can satisfy requirements for all applications and different objectives. Hence, we propose *Memory Cocktail Therapy (MCT)*, a general, learning-based framework that adaptively chooses the best techniques for the current application and objectives.

Specifically, MCT performs four procedures to adapt the techniques to various scenarios. First, MCT formulates a high-dimensional configuration space from all different combinations of techniques. Second, MCT selects primary features from the configuration space with lasso regularization. Third, MCT estimates lifetime, performance and energy consumption using lightweight online predictors (eg. quadratic regression and gradient boosting) and a small set of configurations guided by the selected features. Finally, given the estimation of all configurations, MCT selects the optimal configuration based on the user-defined objectives. As a proof of concept, we test MCT's ability to guarantee different lifetime targets and achieve 95% of maximum performance, while minimizing energy consumption. We find that MCT improves performance by 9.24% and reduces energy by 7.95% compared to the best static configuration. Moreover, the performance of MCT is 94.49% of the ideal configuration with only 5.3% more energy consumption.

CCS CONCEPTS

• **Hardware → Memory and dense storage**;

KEYWORDS

NVM, mellow writes, machine learning, modeling

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3124548

ACM Reference format:
Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T. Chong. 2017. Memory Cocktail Therapy: A General Learning-Based Framework to Optimize Dynamic Tradeoffs in NVMs. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3124548

1 INTRODUCTION

The traditional memory systems based on DRAM technologies have been facing increasing challenges due to DRAM scaling issues. Non-volatile memories (NVM), both commercialized (e.g., NAND Flash [13]) and emerging ones (e.g., PCM [21, 36], ReRAM [49] and NEMS [4, 26]), are considered as promising replacements for DRAM. Compared with DRAM technologies, these NVM technologies offer persistence, much higher scalability and lower stand-by power and non-volatility. However, there are also disadvantages of these NVM technologies. Here are two of the most common ones:

- **Performance.** The write/read access latencies of these technologies are considerably longer than DRAM's. As a result, NVM has lower performance than DRAM.
- **Lifetime.** These NVM technologies usually have limited write endurance [40][51][35][54]. If without special treatment, NVM will have short lifetime because (some of) the memory cells will be worn out soon. As a result, special mechanisms (eg., wear leveling [35] and/or wear limiting [36]) must be used to guarantee the lifetime of NVM memories.

Various techniques have been proposed to combat the performance and lifetime issues of NVMs [5, 51]. Unfortunately, the goals of boosting performance and prolonging lifetime are often in opposition. For example, write cancellation [33][45], which is an effective technique to improve the NVM performance by prioritizing the reads over writes, results in extra writes to the NVM, thus shortening the lifetime. On the contrary, using slower and less destructive writes [51] can improve the NVM lifetime, but at the expense of lower memory system performance.

Intuitively, we can use a combination of techniques (some for lifetime improvement, some for performance improvement) to achieve a sweetspot between performance and lifetime. However, there are several practical issues to achieve this goal:

- **Huge configuration space.** The whole configuration space is huge not only because it may contain multiple techniques, but also because each individual technique itself contains

multiple configurable parameters to control its aggressiveness. For example, in our experiment, the total configuration space includes 3,164 configurations, which is magnitudes larger than the configuration space of prior arts [3, 16, 55].
- **High sensitivity to applications.** The impacts of some techniques are very sensitive to different applications. As a result, the ideal combination of techniques for different applications are dramatically different, as will be shown in Section 3.3. They differ not just in the choices of techniques, but also in the aggressiveness of each chosen technique.
- **User-defined objectives.** Different from several prior proposals that aim at improving a single well-defined goal (e.g., IPC) [3, 16], the desired tradeoff for the NVM changes in different situations. This is especially true when it comes to the lifetime of NVM memory: some users want the memory to last longer (eg. 8 years) and they are willing to pay some performance penalty for it; and some other users want the computing system to run at faster speed even if its NVM system will break down after just few years (eg. 4 years). For these two optimization goals here, the ideal configurations are usually dramatically different.

To adapt the architectural techniques in NVMs to different scenarios, we propose a general, learning-based framework, *Memory Cocktail Therapy (MCT*), which tailors a specific combination of techniques for each application and user-defined objective function. Specifically, MCT formulates all different combinations of techniques into a high-dimensional configuration space and then employs machine learning techniques to model the behaviors of different configurations based on a small set of samples. Given the estimation of all different configurations, MCT selects the optimal configuration that satisfies the requirements specified by the user.

The key challenge in implementing the framework, however, is to select a near-optimal configuration with negligible overhead at runtime, and ideally with little modification to the hardware. We implement MCT so that it automatically reduces the dimensionality of the configuration space by lasso regularization and uses the selected features to guide runtime sampling, leading to more informative samples and higher prediction accuracies. Also, we choose lightweight, yet accurate online predictors such as quadratic regression and gradient boosting, to predict the behaviors of other configurations based on the samples. Furthermore, we implement MCT with a lightweight phase detector and fined-grained sampling technique to accommodate both coarse-grained and fine-grained phases in memory behaviors, which do not rely on prior knowledge about the phases or significant modifications in hardware (except performance counters).

In summary, our work has the following contributions:

- We formulate NVM system design problems with various tradeoffs as constrained optimization problems (e.g. maximizing performance under a lifetime constraint). We motivate the need for machine learning techniques because of the high dimensionality of the configuration space, the high correlation and nonlinear impacts of configurations and the heterogeneity among applications. To our knowledge, our paper is the first to use machine learning techniques to solve such problems.
- Rather than applying machine learning as a black box, we first compare various machine learning models based on their prediction accuracy, computation overhead, convergence rate, etc., and choose the optimal ones. Then, we improve the performance of machine learning models by data normalization, regularization, feature selection, and training with informative sample configurations.
- In addition to machine learning techniques, we use architectural insight to improve our scheme's robustness. For example, we observe that the impact of *Wear Quota* [51] is difficult to predict. Thus, we exclude it from the learning procedures to achieve better accuracy and then use it as the last resort to ensure lifetime goals are met despite inaccurate predictions.
- The framework can also adapt to dramatic phase changes in memory behaviors by guiding the learning procedures with a lightweight phase detector.
- Finally, MCT manages to dynamically choose the near-optimal NVM configuration with no hardware modification and minimal runtime overhead. Compared to a NVM-based system with ideal configurations for different applications, MCT using gradient boosting achieves 94.49% of the maximum performance and consumes only 5.3% more energy.

The rest of this paper is organized as follows. Section 2 introduces NVM architectural techniques and their impacts on performance and lifetime. In Section 3, we provide a case study of the optimization problem in mellow writes. Then we introduce our framework of *Memory Cocktail Therapy* in detail in Section 4 and its implementation in Section 5. The experimental methodology and results are presented in Section 6. Section 7 presents related work, and Section 8 concludes the paper.

2 BACKGROUND

Various trade-offs exist in non-volatile memories which can be utilized to improve their performance or memory lifetime. However, in many cases, the trade-offs used to improve the NVM performance considerably degrade the NVM lifetime, and vice versa.

In this section, we introduce in detail several representative tradeoffs for non-volatile memories and their impact on performance and NVM lifetime. These trade-offs include:

- **With or without Write Cancellation.** Write cancellation [33][45] usually improves NVM performance because it lets read request be served sooner. However, it also degrades memory lifetime since it performs more writes to the NVM memory.
- **With or without Eager/Early Writeback.** Eager writeback [22] utilizes idle memory intervals to eagerly perform write operations of data in the last level cache (LLC) before their eviction, so there is less possibility that the write queue is blocked. As a result, it usually improves performance, However, it also degrades the NVM lifetime since some eagerly written back data need to be rewritten before their eviction.
- **Write Latency VS. Endurance.** The endurance of NVM [51] can be considerably improved if the write operations are performed with lower power and longer latency, thus the NVM lifetime will be significantly prolonged. However,

Table 1: Trade-offs of NVM and Their Impacts on Performance and NVM Lifetime

Trade-offs	Impact on Performance	Impact on Memory Lifetime	Related Proposals
With or without Write Cancellation.	Using write cancellation usually improves performance.	Using write cancellation shorterns NVM lifetime.	[33][45][51][52][37], etc.
With or without Eager/Early Writeback.	Using eager/early writeback usually improves performance.	Using eager/early writeback shorterns NVM lifetime.	[22][32][51],etc.
Write Latency VS. Endurance.	Using long-latency-high-endurance writes degrades performance.	Using long-latency-high-endurance writes prolongs NVM lifetime.	[51][18], etc.
Write Latency VS. Retention.	Using short-latency-short-retention writes usually improves performance.	Using short-latency-short-retention writes shorterns NVM lifetime.	[24][53][23], etc.
Read Latency VS. Read Disturbance.	Using short-latency-high-disturbance reads usually improves performance.	Using short-latency-high-disturbance reads shortens NVM lifetime.	[30][48], etc.

it also significantly degrades the performance of NVM due to longer write latency.

- **Write Latency VS. Retention.** In Multi-Level Cell (MLC) NVM, a write operation usually consists of one RESET and multiple SETs. Shorter write latency can be achieved by reducing the number of SETs in a write operation, at the expense of shorter retention time which requries more frequent refresh writes/scrubs and thus degrades the NVM lifetime [24][53]. Some proposals also claim that there is a similar trade-off in Single-Level Cell (SLC) NVM [23].
- **Read Latency VS. Read Disturbance.** It is also possible to improve the NVM read performance by using short latency but high disturbance reads [30][48]. However, this also comes with NVM lifetime overhead, since such fast reads require frequent refresh/scrub the read NVM cells and thus degrade the NVM lifetime.

Based on these trade-offs, researchers proposed various techniques to improve the performance or lifetime of NVM memory. For example, Eager Writeback [22], Preset [32] and Eager Mellow Writes [51] techniques all utilize the trade-off of Eager/Early Writeback; also, a large amount of proposals (e.g., [52][37][45]) utilize the trade-off of write cancellation. Furthermore, to achieve a performance-lifetime sweet spot of the utilized trade-off, nearly all the proposed techniques (e.g., [33][32][51]) are involved with some configuration mechanism to control its aggressiveness. For each individual application, the ideal configuration of a proposed technique is usually different.

Ideally, if we combine multiple techniques together, the combined technique might achieve a better performance-lifetime balance than all the individual techniques. However, as will be shown in Section 3.1, the configuration space of the combined techniques will be magnitudes larger than the configuration space of each individual technique. As a result, it is challenging to find the ideal configuration among thousands of candidates.

3 CASE STUDY

3.1 Mellow writes configurations

As a case study, we focus on solving the optimization of *Mellow Writes* [51], which covers a series of techniques that utilize multiple tradeoffs in NVMs (eg. *write latency VS. endurance*, *write cancellation* and *eager/early writeback*) . In each technique, there are several configurable parameters that controls the usage of this technique and its aggressiveness. In general, *mellow writes* implements different write latencies and balances between performance and lifetime by carefully scheduling fast writes and slow writes based on the temporal and spatial patterns in the memory system.

- **Default.** *Normal* is the default technique which uses fast (normal) writes. There are two configurable parameters with the default technique: *fast_cancellation* and *fast_latency*. The former one indicates whether to use write cancellation for fast writes, and the latter one is the normalized speed used for fast writes.
- **Bank-Aware Mellow Writes (*bank_aware*).** This technique issues slow writes when the current memory bank is not *busy*. There are three parameters with this option: *slow_cancellation*, *slow_latency* and *bank_aware_threshold*. The first parameter indicates whether to use write cancellation for slow writes. The second parameter is the normalized speed of slow writes. The third parameter controls the aggressiveness of *Bank-Aware Mellow Writes*: when the number of requests to the corresponding bank in the write queue is less than *bank_aware_threshold*, we issue the current write request as a slow write. A higher *bank_aware_threshold* usually results in longer NVM lifetime but lower performance.
- **Eager Mellow Writes (*eager_writebacks*).** This technique eagerly writes back dirty data in the LLC to NVM memory when it is not busy. It has three parameters: *slow_cancellation*, *slow_latency* and *eager_threshold*. The first two parameters are the same with *Bank-Aware Mellow Writes*. The third parameter, *eager_threshold*, controls the aggressiveness of *Eager Mellow Writes*: If the highest N LRU stack positions of last level cache (LLC) contributes less than $\frac{1}{eager\_threshold}$ total hits in LLC, then we consider these N LRU stack positions to be useless and their corresponding LLC dirty entry can be eagerly written back. A higher *eager_threshold* usually corresponds to higher performance but shorter NVM lifetime.
- **Wear Quota (*wear_quota*).** This technique divides the execution into multiple small time slices and assigns a wear quota to each slice. If at the beginning of time slice the accumulated wear quota is reached, the whole coming time slice can only use the slowest writes (in our implementation, 4×) and write cancellation is enforced. The parameter used here is *wear_quota_target* (in years), which indicates the target lifetime of *Wear Quota* technique. A larger *wear_quota_target* enforces longer NVM lifetime, at the expense of a lower system performance.

3.2 Objective tradeoffs

Since NVM systems face multiple constraints (e.g., performance, lifetime and possibly energy [14] as in embedded systems and data centers), the optimization objectives are usually complex and user-defined. In our case, there are three optimization goals: first,

Table 2: Techniques of the evaluated combined technique.

techniques	value	discrete parameters	continuous parameters
Default	N/A	*fast_cancellation*	*fast_latency*
Bank-Aware Mellow Writes (*bank_aware*)	true /false	*slow_cancellation*	*slow_latency* *bank_aware_threshold*
Eager Mellow Writes (*eager_writebacks*)	true /false	*slow_cancellation*	*slow_latency* *eager_threshold*
Wear Quota (*wear_quota*)	true /false		*wear_quota_target*

Table 3: Parameters of the evaluated combined technique.

parameters	value
fast_cancellation	true/false
slow_cancellation	true/false (true if *fast_cancellation* is true)
fast_latency	[1, 4]
slow_latency	[1, 4] (greater than *fast_latency*)
bank_aware_threshold	[1, 4] (in entries per bank)
eager_threshold	[4, 32]
wear_quota_target	[4, 10] (in years)

Table 4: *Ideal* Configurations for different minimal lifetime constraint of application `leslie3d`

	bank _aware	*bank_aware _threshold*	*eager _writebacks*	*eager _threshold*	*wear _quota*	*wear_quota _target*	*fast _latency*	*slow _latency*	*fast _cancellation*	*slow _cancellation*
4.0 years	True	1	True	4	False	N/A	1.0	2.0	False	True
6.0 years	False	N/A	True	4	False	N/A	1.5	2.5	False	True
8.0 years	True	1	True	4	False	N/A	1.5	3.0	False	True
10.0 years	True	4	True	4	False	N/A	1.5	3.5	False	True

Table 5: *Ideal* Configurations for different applications

	bank _aware	*bank_aware _threshold*	*eager _writebacks*	*eager _threshold*	*wear _quota*	*wear_quota _target*	*fast _latency*	*slow _latency*	*fast _cancellation*	*slow _cancellation*
default	False	N/A	False	N/A	False	N/A	1.0	N/A	False	N/A
baseline	True	1	True	32	True	8.0	1.0	3.0	False	True
lbm_ideal	True	4	True	16	True	8.0	1.5	3.0	False	False
zeusmp_ideal	False	N/A	True	8	False	N/A	1.0	1.0	True	False
bwaves_ideal	True	3	True	32	False	N/A	1.0	1.5	False	True
stream_ideal	False	N/A	True	4	True	8.0	1.5	1.5	False	False

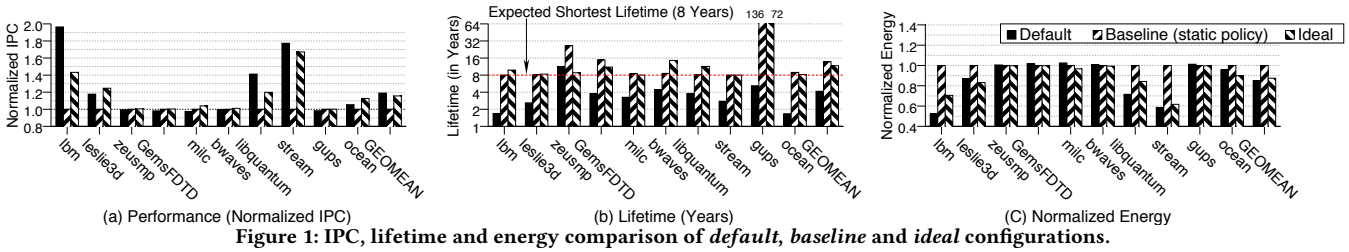


Figure 1: IPC, lifetime and energy comparison of *default, baseline* and *ideal* configurations.

the qualified configurations must guarantee a minimum lifetime; then, our goal is to achieve an IPC as high as possible; finally, among all the qualified configurations whose IPCs are within 95% of the maximum, we choose the one with the lowest system energy as the *optimal* configuration. Let P_i, T_i, E_i represent IPC, lifetime, and energy respectively of configuration i, $\forall i \in (0, N)$. Then the optimization problem can be formalized as follows:

$$\begin{aligned} \min_{i \in (0,N)} \quad & E_i \\ \text{subject to} \quad & T_i \geq t, \\ & P_i \geq 0.95 \times P^* \end{aligned}$$

Although we focus on optimizing IPC and energy efficiency under lifetime constraints in our case study, our framework could be generally applied to optimization problems under other constraints, eg. by switching the three metrics: IPC, lifetime and energy from constraints to objectives and vice versa. For example, in embedded systems, the objectives could be to enforce a constraint on energy, while maximizing performance and lifetime. In data centers, however, the objectives could be to guarantee a performance target, while maximizing lifetime and minimizing energy. Our learning framework is flexible with user-defined objective functions since the main challenge we want to solve is to model performance, lifetime and energy of all different configurations based on a small number of samples, yet at negligible cost at runtime.

3.3 Challenges

The optimization problem (i.e., choosing the *optimal* configuration) is trivial if we have the data for all configurations: $[P_i, T_i, E_i]$, $\forall i \in (0, N)$. However, in this section, we will show that it is impractical to do so for three reasons: huge configuration space, high sensitivity to applications and high sensitivity to user defined objectives.

3.3.1 Huge Configuration Space. As *mellow writes* covers a series of techniques and multiple tradeoffs, the whole configuration space is non-trivial, as is shown in Tables 2&3. To reduce its size, we add three constraints to remove impractical configurations:

- The *parameters* are used only when it is enabled by the selected techniques. For example, when *eager_writebacks* technique is not selected, *eager_threshold* is meaningless and thus not considered.
- The *slow_latency* must be larger than *fast_latency*.
- When *fast_cancellation* is true, we force *slow_cancellation* to be true. Prior work [51] shows that write cancellation for slow writes is more effective in improving performance compared with write cancellation for fast writes. Therefore, it does not make sense to have a technique which offers write cancellation for fast writes, but not for slow writes.

However, even with these constraints, the configuration space of *Mellow Writes* is still huge—there are 3,164 different configurations in total. Brute-force search of the whole configuration space is very expensive. In our experiments, in order to compare our framework with the ideal configuration, we simulate all the configurations for 10 applications , which consumed more than 300,000 computing hours. The high evaluation cost makes the selection of the optimal configuration at runtime quite challenging.

3.3.2 High Sensitivity to User Defined Objective. The choice of optimal configuration is highly affected by the user defined objective. For application `leslie3d`, we vary the minimal lifetime requirement from 4 years to 10 years. The results are shown in Table 4. Due to the experiment time constraint, we explored a limited configuration space without the usage of *Wear Quota* for this table, but the results clearly indicate that the optimal configuration varies with different user defined objectives.

3.3.3 High Sensitivity to Applications. Not only does the optimization objective affect the choice of optimal configuration, but also the currently executed application. Figure 1 shows the optimal under the default optimization objective (i.e., 8-year lifetime, an IPC within 95% of the maximum IPC while minimizing energy). For comparison purposes, we also have two representative configurations: *default*, which does not use any mellow writes techniques; and *baseline*, which is the static configuration used in prior work [51].

We can see that, the effectiveness of *baseline* configuration is far from ideal—for more than half of the applications, the performance of *baseline* is significantly lower than *ideal*. However, as is shown in Table 5, finding out the *ideal* configurations is certainly not an easy task. In fact, in all the ten evaluated applications, none of them share the same *ideal* configuration.

4 MEMORY COCKTAIL THERAPY

To address the challenges and solve the constrained optimization problem as discussed in the case study, we propose *Memory Cocktail Therapy*, a general, learning-based framework to dynamically model IPC, lifetime and system energy of different combinations of techniques for each application. We first formalize the problem in Section 4.1 and then quantitatively analyze the problem complexity in Section 4.2. The high complexity of our problem space indicates that statistical modeling is necessary, therefore we investigate various learning algorithms and evaluate their performance and computational overhead in Section 4.3. Finally, in Section 4.4 we further improve the accuracy of selected learning algorithms based on the insights from both machine learning and computer architecture perspectives.

4.1 Problem Formalization

We formulate the configuration space and tradeoff space to have a well-defined interface (inputs and outputs) for learning and optimization procedures.

4.1.1 Configuration Space. We represent all configurations with 10-dimensional vectors:

$$x = \begin{bmatrix} bank\_aware \\ bank\_aware\_threshold \\ eager\_writebacks \\ eager\_threshold \\ wear\_quota \\ wear\_quota\_target \\ fast\_latency \\ slow\_latency \\ fast\_cancellation \\ slow\_cancellation \end{bmatrix} \quad (1)$$

Application	Top-3 most effective features
lbm	$-fast\_latency$, $+fast\_latency^2$, $+slow\_cancellation^2$
leslie3d	$+slow\_cancellation^2$, $-eager\_writebacks * slow\_cancellation$, $+eager\_writebacks * fast\_latency$
GemsFDTD	$+slow\_cancellation^2$, $-slow\_latency * slow\_cancellation$, $+slow\_latency$
stream	$-fast\_latency$, $+slow\_cancellation^2$, $-eager\_writebacks * fast\_latency$

Table 6: Most effective quadratic features in different applications

For example, the following vector, $[1, 1, 1, 32, 0, 0, 1.5, 3.0, 0, 1]^T$ represents a combination of techniques that uses bank-aware mellow writes with a threshold of 1, eager writebacks with an eager threshold of 32, fast and slow write latencies of 1.5x and 3.0x, and write cancellation only on slow writes.

4.1.2 Tradeoff space. We include three metrics in the objective space: IPC, lifetime and system energy, as discussed in 3.2. Therefore, we formulate the tradeoff space into 3-dimensional vectors:

$$y = \begin{bmatrix} IPC \\ lifetime \\ system\_energy \end{bmatrix} \quad (2)$$

4.2 Quantitative analysis of the problem space

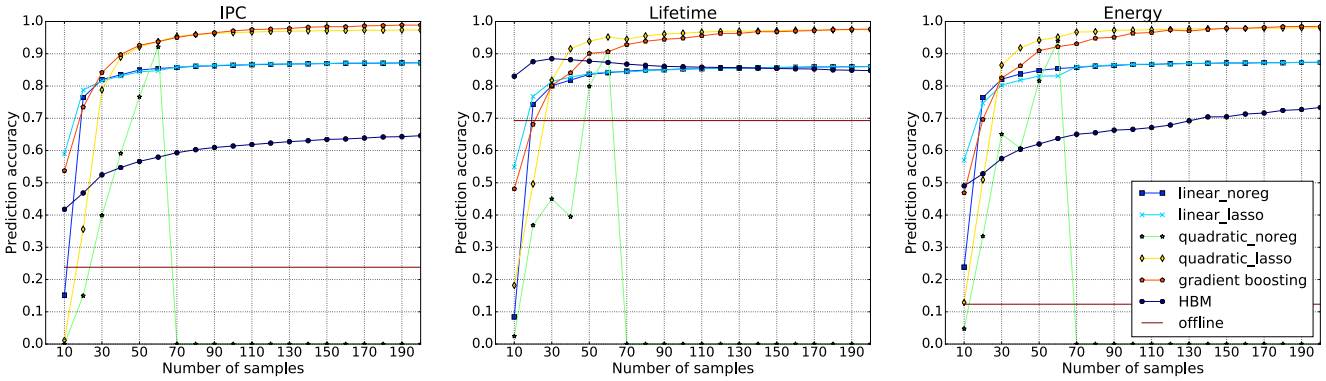
To demonstrate the problem complexity, we quantify the impacts of different input parameters and their correlation. We fit the training data to a quadratic regression model with lasso regularization, which show high prediction accuracy in later experiments. More details about the model will be introduced in Section 4.3. In this model, the input parameters are extended to quadratic features including both single knobs and knob pairs. The post-training weights of these features indicate their effectiveness and impacts on the outputs.

Then, we rank the top-3 most effective features for different applications, as shown in Table 6. From the effectiveness ranking results, we can see that: 1) some of these top-ranked features are knob pairs, which indicates high correlation between input parameters and the importance of their correlation. Thus, it is necessary to model the joint contribution of these parameters to the outputs. 2) different applications have entirely different top-ranked knobs/knob-pairs, so it is difficult to determine the effectiveness order statically. 3) single knobs can have nonlinear impacts on the outputs (e.g. *fast_latency* on *lbm*). Note also, we have a minimum lifetime constraint in the optimization problem. As a result, it is difficult to determine the best value for each knob to satisfy the lifetime constraint while maximizing performance and energy efficiency without statistical modeling.

Therefore, we exploit machine learning techniques to model the relationship between configurations and the outputs, as discussed in the next section.

4.3 Learning performance, lifetime and energy

To model performance, lifetime and energy, we investigate three learning algorithms: 1) *Regression*, 2) *Boosting algorithms*, 3) *Hierarchical Bayesian models*.



(a) IPC **(b) Lifetime** **(c) Energy**

Figure 2: Convergence rates and prediction accuracies using different models

Regression models the functional relationship between configuration vectors and objective vectors, based on a small set of sample configurations. One drawback of regression is overfitting, which can happen when the model has many parameters (e.g., higher-order polynomials). Regularization reduces model complexity to avoid overfitting. Lasso, the least absolute shrinkage and selection operator, is a common regularization technique [47]. We will show that by using lasso, we can speed up the convergence of regression-based predictors (in Figure 2), as well as guide the feature selection for more informative runtime sampling (in Figure 4).

Boosting algorithms are a class of ensemble learning methods combining multiple weak learners into a strong learner [39]. Gradient boosting is a state-of-art boosting algorithm for learning regression models [10, 27]. It is also one of the most accurate methods for our data sets as shown later.

Hierarchical Bayesian models do not learn the functional relationship between inputs and outputs, but assume that some latent variables are shared between different applications and learning their posterior distributions allows predictions for the current application [29]. Rather than overfit, this model uses only similar applications to predict new application behavior; however, accuracy requires that the training set has sufficient breadth to find known applications that correlate with the new application.

4.3.1 Model selection. Table 7 and Figure 2 compare these models in terms of (1) requirements for online or offline data, (2) computation overhead (in microseconds), (3) convergence rates (in samples), and (4) prediction accuracies. We use *coefficient of determination* as our accuracy metric:

$$acc = \max\left(0, \left(1 - \frac{\|(Y' - Y)\|_2^2}{\|(Y - \overline{Y})\|_2^2}\right)\right) \quad (3)$$

where Y' represents the prediction, Y represents the true data and $\overline{Y}$ represents the mean of the true data. The coefficient of determination measures the proportion of the variance in dependent variables that can be predicted by independent variables; a commonly used metric to evaluate prediction accuracy [44]. The computation overhead is tested on a 12-core Intel i7 processor with 64 GB memory.

The *offline* predictor averages data from training applications to predict the current application. There is no runtime overhead, but prediction accuracy is low. The *Hierarchical Bayesian model* has high prediction accuracy and fast convergence rate on lifetime because of the high correlation among benchmark applications. However, the IPC and energy predictions are not accurate because the data magnitudes vary significantly among different applications. The biggest issue with this model, however, is that it takes 8,000 ms to produce a prediction. We will not focus on this predictor in this paper. However, for future work, it is possible to design specialized hardware for the model to mitigate the runtime overhead. Among online models, *gradient boosting* and *quadratic regression with lasso regularization* both achieve high prediction accuracies on all three objectives and have reasonable runtime cost. *Quadratic model without lasso regularization* has difficulty converging before 200 samples. The reason is that the input vectors are expanded from 10 dimensions to 65 dimensions in the quadratic model, including square terms, cross terms and linear terms. Without regularization, the model is prone to overfitting given small sample size. *Linear models* are not as accurate as quadratic models, which indicates that the underlying features have complicated, interdependent relationships to the targets.

Predictors	Need offline data?	Need online data?	Computation overhead
offline	Yes	No	0 ms
linear model, no regularization	No	Yes	1 ms
linear model, lasso regularization	No	Yes	1 ms
quadratic model, no regularization	No	Yes	3 ms
quadratic model, lasso regularization	No	Yes	8 ms
gradient boosting	No	Yes	112 ms
hierarchical Bayesian model	Yes	Yes	8,000 ms

Table 7: Comparison of different models

According to Table 7 and Figure 2, we choose gradient boosting and quadratic regression with lasso in our final experiments because they achieve high prediction accuracies with low computation overhead.

4.4 Improving Prediction Accuracies

Rather than applying existing learning techniques as black boxes, we can make small modifications to improve their behavior on the particular problem of optimizing non-volatile memory configurations. We find that prediction accuracy is improved using the following techniques:

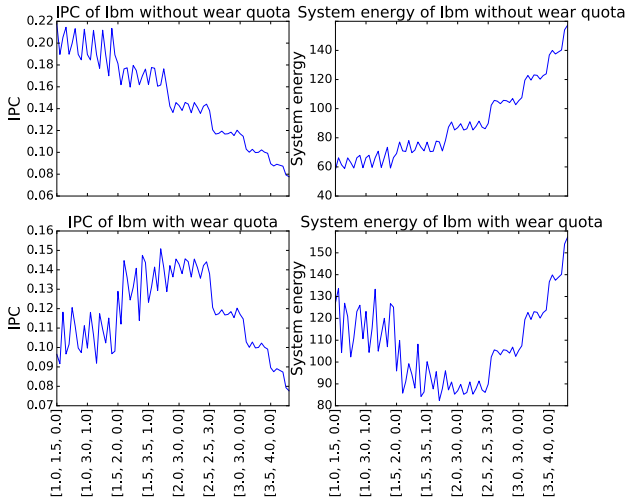
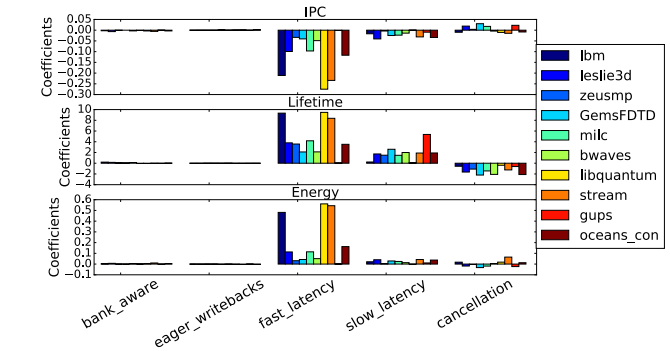


Figure 3: Including or excluding wear quota in the configuration space. We take benchmark *lbm* from SPEC CPU 2006 as an example and choose 77 sample configurations using feature-based sampling. The xlabels are in the format of: [*fast_rate, slow_rate, write_cancellation* (on slow and/or fast writes)]. From left to right, the configurations have increasing write latencies. Including wear quota adds more complexity to the modeling and degrades prediction accuracies by 2% ∼ 6%**.**

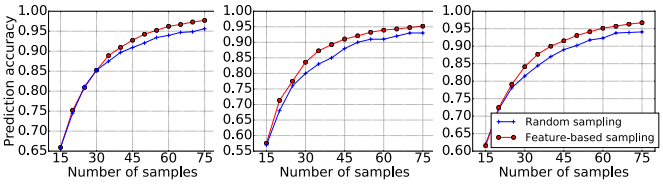
Normalization avoids extreme coefficient values for different parameters. We normalize all the data to the baseline configuration's measured behavior. This technique improves both the prediction accuracy and convergence rate; however, the predictor now only learns how different a configuration is from the baseline. Thus, we periodically run the baseline configuration and multiply the prediction data by the baseline data. This method accommodates small system phase changes and avoids oscillation by using absolute values.

Including or excluding wear quota. We observe that *wear quota* is a technique that makes significant impact on the three objectives. Wear quota is a technique to guarantee a minimum lifetime[51]. It does so by ensuring all memory writes are slow when the fast-write quota for a period is exceeded. Figure 3 illustrates the impacts of wear quota. Compared with the data not using wear quota, the data using wear quota exhibit higher complexity. Their performance and energy efficiency is hard to predict when the write latencies are either very low or very high. When write latencies are very low, wear quota is triggered, resulting in slow writes. When write latencies are very high, the memory system is intrinsically slow and its performance and energy efficiency is bad. We observe 2% ∼ 6% degradation in prediction performance when including data using wear quota. Therefore, we exclude wear quota from our configuration space for prediction. However, we use it as a *fixup* technique later to guarantee the lifetime target for configurations whose lifetime were overestimated during prediction.

Feature selection. In the previous discussion, we use random sampling to compare different predictors. However, we find that the prediction accuracy increases if we sample based on prior knowledge about what features are most important in the configuration space.



(a) Coefficients of lasso regression with linear model. Only important input features have nonzero coefficients.



(b) IPC **(c) Lifetime** **(d) Energy**

Figure 4: Feature-based sampling vs. random sampling on gradient boosting.

To find important features, we first manually cluster the 8 features (excluding wear quota) based on domain knowledge. For example, *bank_aware* and *bank_aware_threshold* are merged into one variable, *bank_aware*, that has 5 levels from 0 to 4. Following this approach, we compress the original 8 features into 5 features: *bank_aware*, *eager_writebacks*, *fast_latency*, *slow_latency* and *cancellation*. This manual compression further aids dimensionality reduction. The configuration space, however, requires both the usage and the aggressiveness parameter for each technique.

Then we use lasso regularization to identify important features. Figure 4a shows the coefficients of lasso regression with a linear model. The coefficients of *bank_aware* and *eager_writebacks* are near zero for all objectives of all applications. The three important features are thus: *fast_latency*, *slow_latency*, and *cancellation*. The same features in square terms and cross terms are also the important features in the quadratic model with lasso regularization.

Based on the selected features, we obtain 77 samples by uniformly sampling from the three primary features and randomly sampling from the left. The feature-based sampling leads to higher prediction accuracies for all other configurations. For example, the performance of gradient boosting increases by about 3% on average for all objectives.

Additionally, the three primary features indicate that our framework is not limited to the techniques in this paper. It could be generally applied to architectural techniques in NVMs that involve these three features [23, 32, 53].

5 IMPLEMENTATION

In this section, we discuss how to implement the learning-based framework in practical NVM systems. There are several challenges in the implementation. First, in real memory systems, there are various access patterns along time and among different applications. Meanwhile, the performance of architectural techniques varies

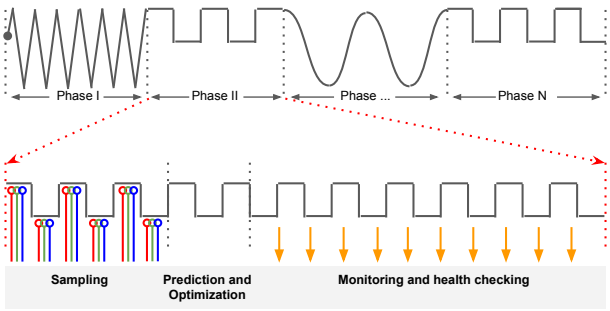


Figure 5: System implementation

significantly for different access patterns as they exploit different tradeoffs. Thus, we assume that the sample configurations and the chosen optimal configuration should run on similar memory behaviors. Otherwise, the information we learned from the training is not valuable.Second, real memory systems usually exhibit fine-grained bursty behaviors. However, all the different sample configurations in our framework should run on similar memory workload. We should schedule these samples carefully to avoid discrepancies in their workload. Third, although we have selected predictors with accuracies higher than 90%, there still exist mispredictions. Nevertheless, the final optimal configuration should still satisfy the hard constraint in the user-defined objective function, eg. the minimum lifetime target. Finally, we should guarantee that after optimization, our system will not perform worse than the baseline.

To address all these issues, we exploit a series of techniques, including phase detection, fine-grained runtime sampling, wear-quota fixup and periodic health checking. Figure 5 illustrates the workflow of our framework. And we discuss each technique in detail in the following subsections.

5.1 Phase detection

Phase detection methodologies have been widely investigated in system area [8, 11, 29, 41–43]. Since memory systems exhibit frequent, fine-grained phase changes, our goal is to implement a light-weight phase detector that recognizes only dramatic changes in memory behaviors. Our system could tolerate minor phase changes by normalization, as discussed in Section 4.4, and fine-grained runtime sampling, as discussed in Section 5.2. Therefore, we adapt the phase detection methodology in [29] and use Student's t-test to emphasize the detection of dramatic phases. Our phase detection algorithm is as follows:

- Use performance counters to monitor the memory workload (including both read requests and write requests) for every I instructions.
- Keep a history record of the memory workload during the past $1000 * I$ instructions.
- Perform the two-sided Student's t-test for the last $100 * I$ instructions and the past $1000 * I$ instructions based on the mean values and standard deviations of their memory workload. The t-test score indicates the confidence in rejecting the hypothesis that the mean values of the memory workload in the two testing windows are the same. The higher the t-test score is, the more confident we are to recognize a new phase with respect to the memory behaviors.

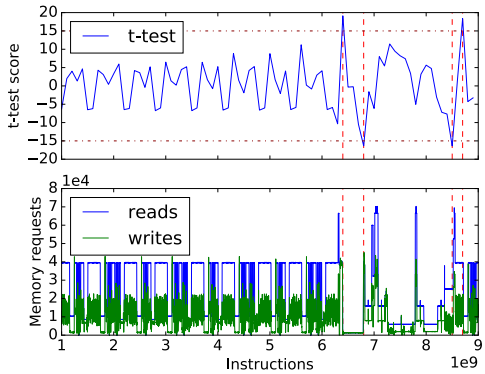


Figure 6: Phase detection. (I is 1,000,000 instructions and $score_{threshold}$ is 15.) Recognized phases in memory behaviors are marked with red vertical lines, which correspond with the changes in t-test scores.

- When the t-test score exceeds a threshold, $score_{threshold}$, we recognize a new phase. Then we clear off the counters and restart.

We demonstrate the phase detection result of *ocean* in Figure 6. Our phase detector can accurately capture coarse-grained phases, while tolerate frequent, fine-grained phases.

Note that, higher accuracy might be achieved by providing more program information to the phase detector. This can be done through either hardware modification (e.g., passing PC with memory requests) or application/compiler modification (e.g., generating compiler hints [8, 11, 41, 43]). In this paper, the recognized phases shown in Figure 6 are sufficient to guide the learning procedures. Meanwhile, our phase detection scheme only relies on the statistics provided by existing performance counters.

5.2 Runtime sampling

Memory bursty behaviors are common in memory intensive applications (eg. *lbm*, *libquantum*, *milc*, etc.) [17]. If we evenly divide the sampling period into N slices for N samples, then some samples may fall in bursty periods while others fall in idle periods. To accommodate these fine-grained memory patterns, we propose fine-grained sampling:

- Assume the total sampling period covers T instructions.
- Run each sample configuration for a small sampling unit, which covers t instructions.
- Loop over all samples for $\frac{T}{N*t}$ times.
- Accumulate statistics of each sample configuration during $\frac{T}{N*t}$ sampling units.

We observe that the magnitude of memory burst length is at least 10 million instructions in our benchmarks. Hence, all the sample configurations could be scheduled within each memory burst if we use a small sampling unit, eg. 100 kilo instructions. Note also, the configuration of sampling unit length and the number of iterations can be guided by the mean memory workload in the current phase, as discussed in Sec 5.1. If the mean value is low, we can use large sampling units so that each sample configuration can make effects with enough memory requests. Otherwise, we can use small units and repeat many times so that all configurations can evenly experience all memory patterns.

The fine-grained sampling methodology is lightweight so that it incurs negligible overhead at runtime. With cyclic-fine-grained sampling, each sample will cover a wide set of memory behaviors. And all different samples are exercised on similar memory behaviors. Using the fine-grained sampling methodology together with the phase detector, our system can accommodate both fine-grained and coarse-grained phases in the memory system.

5.3 Prediction and optimization

After the sampling period, we collect the IPC, lifetime and energy data of the sample configurations. Then we apply the learning algorithms to predict the three objectives of all other configurations. After the prediction, we choose the optimal configuration based on the used-defined objective function, as defined in Section 3.2.

However, the optimization based on prediction data may have errors. For example, the chosen optimal configuration might be overestimated in lifetime while its real lifetime cannot satisfy the minimum target. To correct such prediction errors, we add *Wear Quota* to the chosen optimal configuration and set the *Wear Quota* target to the minimum lifetime target. Then the minimum lifetime could be guaranteed. If the chosen optimal configuration could meet the lifetime requirement itself, adding *Wear Quota* incurs negligible overhead according to the prior work. For prediction errors in IPC and energy, we account that in our final optimization results and compare them with the baseline and ideal policy.

5.4 Monitoring and health checking

After choosing the optimal configuration, we can launch the optimal configuration in the memory system. However, the chosen optimal configuration might be suboptimal in practice due to prediction errors. Hence, we monitor the performance of the memory system and periodically switch back to the baseline configuration for health checking. During this stage, we can perform several health checks of the current system. First, we can obtain the memory workload statistics for the phase detection. Second, we can check minor phase changes by just monitoring the statistics of the baseline configuration. With the normalization technique, periodic statistics of the baseline configurations can help adapt to minor phase changes and avoid system oscillations. Third, if we find that the performance of the chosen configuration is worse than the baseline configuration, we can switch to the baseline configuration so that our system will never be worse than the baseline system.

6 RESULTS

6.1 Experiment setup

We use gem5[2] and NVMain[31], which is a timing-accurate simulator for non-volatile memories. Table 8 and Table 9 respectively report the detailed parameters of processor and resistive memory based main memory. We assume the memory system uses an efficient bank-level wear-leveling technique. Without loss of generality, we use ReRAM[50] for the simulated NVM-based main memory. According to recent represented commercial products[38], we model the baseline write latency to be 150ns and its endurance as 8×10^6. We also model the *Write Latency VS. Endurance* trade-off [51] in our simulation framework: the write latency can be extended to $150 * wr\_ratio$ ns, as a result, the write endurance can be improved quadratically to $8 * 10^6 * wr\_ratio^2$. For processor and memory systems, we respectively use MCPAT [25] and NVSim [6] to model their energy consumption.

Table 8: Processor Simulation Parameters

Freq.	2GHz
Core	Alpha ISA, single-core, OoO, 64-byte cacheline, 8-issue
L1$	split 32KB I/D-caches, 4-way, 2-cycle hit latency, 8-MSHR
L2$	256KB per core, 8-way, 12-cycle hit latency, 12-MSHR
L3$(LLC)	2MB, 16-way, 35-cycle hit latency, 32-MSHR

Table 9: Main Memory System Simulation Parameters

Basics	400 MHz, 4GB, 64-bit bus width, using ReRAM, assume using effective wear-leveling scheme (e.g., Start-Gap [35]) in bank granularity which can achieve 95% average lifetime, write-through (writes bypass row buffers), 1KB row buffer, open page policy, tFAW=50ns
# of Banks	16
# of Rows	8192 per bank
# of Cols	512 per row
Read Queue	64 entries, highest priority
Write Queue	64 entries, middle-high priority write drain threshold: 32 (low), 64 (high)
Eager Mellow Write Queue	32 entries per channel, lowest priority, no write drain, slow writes
tRCD	48 cycles (120 ns)
tWP (WR pulse lat.)	$60 * wr\_ratio$ cycles ($150 * wr\_ratio$ ns).
tCAS	1 cycle (2.5 ns)
endurance	$8 * 10^6 * wr\_ratio^2$ writes;

We run 7 memory-intensive workloads (*lbm, leslie3d, zeusmp, GemsFDTD, milc, bwaves* and *libquantum*) from SPEC CPU2006, *ocean* from SPLASH-2, and 2 extra microbenchmarks (*gups* and *stream*) which separately provides random and stream memory access pattern. Each benchmark is warmed up for 6 billion instructions and simulated in detail for another 2 billion instructions. To calculate lifetime, we assume the system will cyclically execute the current workload until the main memory wears out, and this total execution time is the memory lifetime for the workload.

6.2 Evaluation of memory cocktail therapy

In this section, we evaluate *MCT* with respect to our three objectives: *IPC*, *lifetime* and *system energy*. We compare *MCT* with three baseline systems: 1) *default*, which does not use mellow writes techniques, 2) *best static policy* chosen from prior work, which uses bank aware mellow writes and eager writebacks with slow write cancellation and wear quota. 3) *ideal policy*, which is selected by a brute-force search through the whole configuration space. We demonstrate *MCT* using gradient boosting and quadratic lasso as the learning models without using wear quota in the learning process. After the prediction, *MCT* uses wear quota as a fixup technique for the predicted optimal configuration to guarantee a minimum lifetime target.

6.2.1 Better tradeoffs between IPC, lifetime and system energy. Figure 7 shows the comparison of *MCT* with other baseline systems. We assume that the lifetime requirement is 8 years here. And we normalize the IPC and energy data by the *best static policy*. The *default* system only uses fast writes so it achieves high IPC and low energy consumption. However, it cannot satisfy the minimum lifetime requirement for most of our benchmarks (except *zeusmp*). Both *MCT* and the *best static policy* can guarantee the minimum lifetime requirement, while *MCT* using gradient boosting achieves 9.24% higher IPC and 7.95% lower energy consumption on average for all the benchmarks. Particularly, *MCT* achieves significantly better

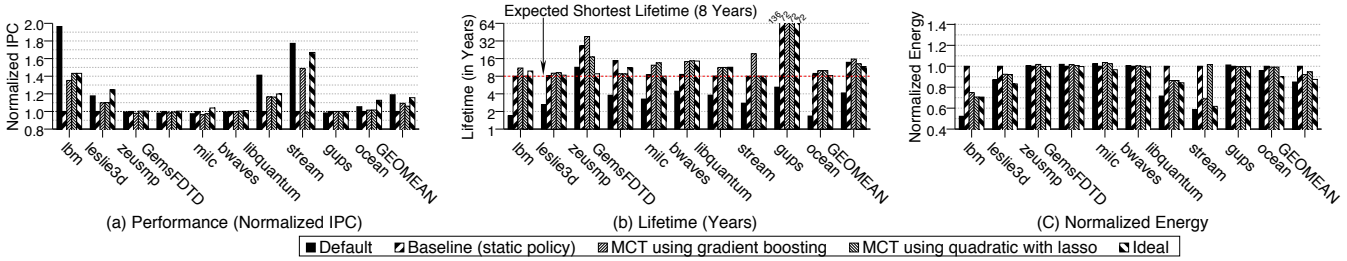
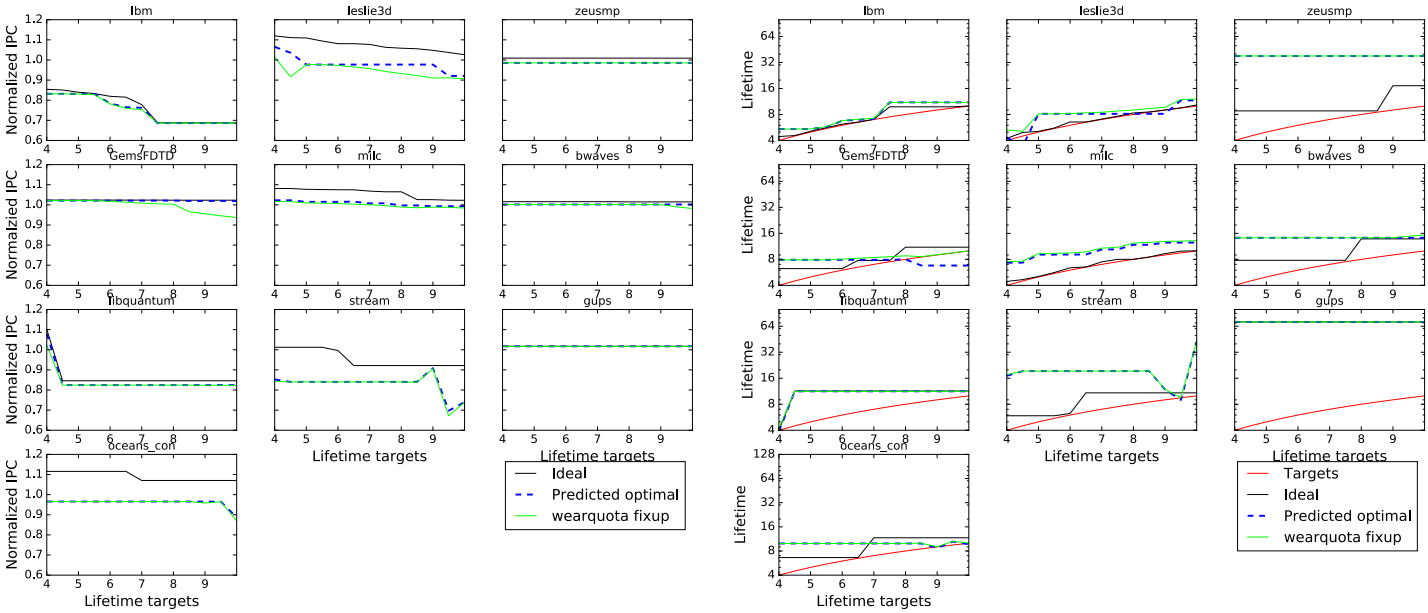


Figure 7: Compare *Memory Cocktail Therapy* with baseline systems with respect to IPC, lifetime and system energy.

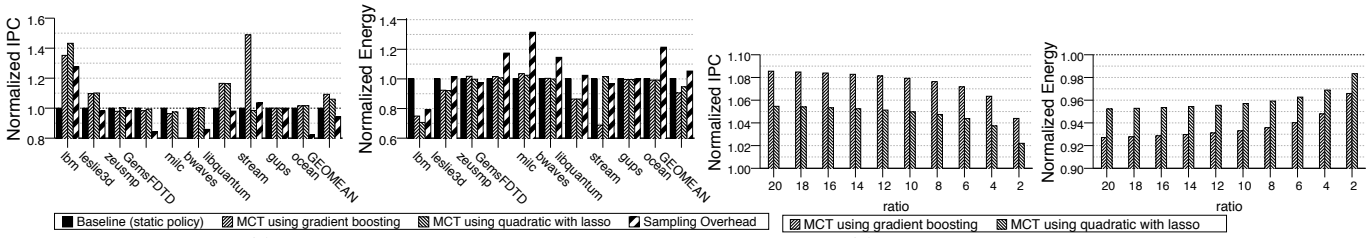
	bank _aware	*bank_aware _threshold*	*eager _writebacks*	*eager _threshold*	*wear _quota*	*wear_quota _target*	*fast _latency*	*slow _latency*	*fast _cancellation*	*slow _cancellation*
static	True	1	True	32	True	8	1.0	3.0	False	True
lbm	True	1	True	32	True	8	2.0	2.5	False	False
libquantum	True	3	True	32	True	8	1.5	3.0	False	True
stream	False	N/A	True	32	True	8	1.5	2.5	False	False
ocean	True	2	True	4	True	8	1.5	3.5	False	True
bwaves	True	4	False	N/A	True	8	1.0	1.5	False	True

Table 10: Optimal configurations for different applications selected by MCT with gradient boosting.



(a) IPC **(b) Lifetime**

Figure 8: Sensivity to different lifetime targets



(a) *Gains vs. losses* **(b) Extrapolate gains and losses**

Figure 9: Sampling overhead

tradeoffs when the *best static policy* is far behind the *ideal policy*, as in *lbm*, *leslie3d*, *libquantum*, and *stream*. For other benchmarks that the static policy is already close to the ideal policy, *MCT* performs similarly to the baseline with negligible degradation in performance and energy efficiency. We list the optimal configurations selected by MCT in Table 10. Small variations in the configuration can result in significant improvement over the static policy. For example, in *lbm*, the only differences between the MCT-optimal configuration and the static configuration are in *fast_latency*, *slow_latency*, and *slow_cancellation*. However, the performance improvement is

up to 35%. The main reason is that, the high *slow_latency* together with *slow_cancellation* in the static policy causes many rewrites, which hurts lifetime. Eventually, more wear quota slow writes are enforced in order to guarantee the minimum lifetime target of 8 years.

Compared with the *ideal policy*, *MCT* using gradient boosting achieves 94.49% of the maximum performance with only 5.3% more energy consumption. *MCT* using quadratic with lasso achieves 6% performance gains and 5.3% energy savings compared with the static policy. And it achieves 91.69% of the ideal performance with 8.3% more energy. In particular, it performs well on most of the benchmarks except *stream*. This indicates that quadratic model may not work well for every application depending on the intrinsic memory characteristics while gradient boosting is more general as it includes a variety of weak predictors.

6.2.2 Sensitivity to different lifetime targets. Figure 8 shows the results of *MCT* with different lifetime targets. We only show MCT using gradient boosting here. In our experiments, the lifetime targets range from 4 years to 10 years. In general, when we have higher lifetime targets, the chosen optimal configuration has lower performance and higher energy consumption, and vice versa. *MCT* can generally capture this trend. However, there are discontinuities in the predictions, as well as the ideal policies for different lifetime targets. One reason is that, the set of configurations we experiment on is still far from complete compared with the oracle, although it already contains 3,164 configurations and needs more than 300,000 computing hours to simulate for all benchmarks. Nevertheless, MCT still manages to select a better configuration than the baseline configuration as in *lbm, stream, lelsie3d*, etc. while saving large amount of evaluation time for each configuration. When the chosen optimal configuration has overestimated lifetime (eg. in *GemsFDTD* when the lifetime target is beyond 8 years), the wear-quota fixup technique works as the last resort to guarantee that the minimum lifetime requirement will still be satisfied.

6.2.3 Reduced sampling and learning complexity by excluding wear quota in the learning process. Although for some benchmarks (eg. *lbm* and *stream* as shown in Table 5), including wear quota in the learning process leads to better choices, there are two practical problems with wear quota prediction: 1) The prediction accuracies including wear quota degrades by 2% ∼ 6% for all applications, as discussed in Section 4.4. 2) Wear quota is a technique that depends on aggregate memory behaviors in a long period. However, we need short sampling period to mitigate runtime sampling overhead and fine sampling granularity to tolerate memory bursty behaviors. Including wear quota in the prediction space adds more challenges in practical sampling methodologies. For example, using the same fine-grained sampling methodology, the chosen optimal configuration for *lbm* including wear quota only achieves 70% of the performance by excluding wear quota, while consuming 50% more energy. And we observe similar phenomenon in *leslie3d*, which has 6% performance degradation with 13% more energy consumption when including wear quota in the prediction.

6.2.4 Sampling overhead. During the sampling period, MCT exercises different configurations in order to model IPC, lifetime and energy for all configurations. However, these sample configurations

Table 11: Multi-program workloads

mix1	lbm, libquantum, stream, ocean
mix2	leslie3d, bwaves, stream, ocean
mix3	GemsFDTD, milc, zeusmp, bwaves
mix4	lbm, leslie3d, zeusmp, GemsFDTD
mix5	GemsFDTD, milc, bwaves, libquantum
mix6	libquantum, bwaves, stream, ocean

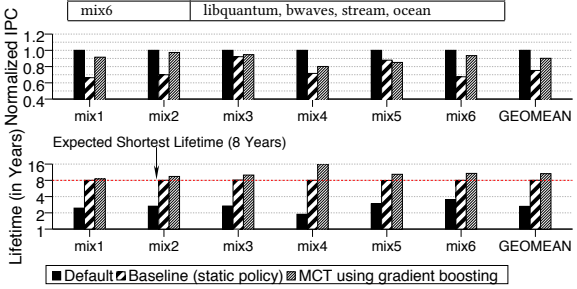


Figure 10: *Memory Cocktail Therapy* in multi-core environments.

are generally not the optimal one. In this section, we evaluate the overhead caused by running suboptimal configurations and compare the gains in testing period to the losses in the sampling period. We refer the rest of the phase as the sampling period to testing period here.

Figure 9a shows the comparison. All the data are normalized by the static policy. On average, the aggregate IPC of sample configurations is 94.32% of the baseline, while MCT using gradient boosting achieves 1.09x IPC of the baseline. Similarly, the aggregate energy consumption of sample configurations is 1.05x of the baseline, while the predicted optimal configuration only consumes 92.05% energy of the baseline.

Considering the overhead during the sampling period, the practical gains of MCT depend on the ratio between the testing period length and the sampling period length. As a proof of concept, in our experiments, the sampling period covers 1 billion instructions and the testing period covers 2 billion instructions. However, according to prior work on memory workload characterization of SPEC CPU 2006 benchmarks[17], there are only a few phases during the whole program execution. And each program has thousands of billions instructions. To have a better idea about the practical gains, we extrapolate the overhead and gains with different ratios between the testing period length and the sampling period length. If the testing period is α times of the sampling period, then the extrapolated IPC is:

$$IPC_{total} = (IPC_{sampling} + \alpha * IPC_{testing})/(1 + \alpha) \quad (4)$$

Figure 9b shows the total IPC and energy consumption by extrapolating gains and losses based on Equation (4). For example, if the testing period is 10x of the sampling period, which is a reasonable case from the characterization results[17], then *MCT* using gradient boosting can still achieve 7.93% performance gains and 6.7% energy savings compared with the static policy.

6.2.5 Extension to multi-program workloads. We also investigate the effectiveness of MCT in a multi-core architecture. The architecture has 4 cores, independent L1/L2 cache for each core, a shared 8MB L3 cache and an 8GB, 32-bank resistive main memory. We randomly pick 4 benchmarks and execute them concurrently, and then evaluate the performance (in normalized geometric mean IPC) and lifetime (in years). The multi-program workloads that we experimented on are listed in Table 11. Our results demonstrate that MCT also performs well in multi-program situations, as shown

in Figure 10. Compared to the static policy, MCT achieves around 20% (geometric mean) performance benefits and also satisfies the 8-year lifetime requirement. We only compare MCT with the static policy for single-core architectures because exploring the design space in multi-core architectures for comparison is computationally intractable. Meanwhile, MCT can automatically find a good configuration for the current multi-program workload by only exercising a small set of sample configurations at runtime.

We found that, multi-core workloads tend to smooth phase behavior out as soon as they have a least a few programs. Similar results have also been reported in the literature [28]. That is why MCT also performs well in multi-core environments. The only dramatic effect, however, is when one or more programs start/exit. In such situations, our phase detection scheme is still necessary to provide rapid adaptation to this large shift in memory behavior. The effectiveness and fairness of MCT can be further improved by exploiting multi-program specific characteristics (e.g., by utilizing the schemes similar to [9, 34]) and we leave it as our future work.

7 RELATED WORK

Machine learning approaches have attracted more interest recently to assist the architecture design as systems and architectures are becoming more complicated. These techniques have been used for automatic resource allocation [3, 16], branch prediction/LLC reuse-distance prediction [12, 19, 46], performance modeling [1, 7, 20], etc. For example, ensembles of Artificial Neural Networks (ANNs) were exploited to coordinate the allocation of multiple interacting resources to different applications and thus optimize the system-level performance [3]. Reinforcement learning (RL) was used to adapt the memory scheduling policies to changing workload and maximize the memory utilization [16]. Markov Decision Process (MDP) was used to model how the RL-based memory controller interacted with the rest of system.

Particularly, machine learning approaches can provide guidance on how to determine the optimal architectural configuration for specific applications and targets. LEO [29] exploited a hierarchical Bayesian model to learn the system behaviors of various processor and memory configurations and then minimize energy under performance constraints. JouleGuard [15] employed a combination of machine learning (eg. multi-armed bandits) and control theoretic techniques (eg. PI controller) to provide energy guarantees while maximizing accuracy. CASH [55] also exploited a combination of control theory and machine learning techniques to learn fine-grained configurations of multi-core architectures in IaaS Clouds to provide near-optimal cost savings for different QoS targets.

Our proposal extends the application of machine learning approaches to emerging memory technologies, which introduced a new constraint in the optimization space: lifetime. The lifetime constraint increases optimization complexity and leads to significantly different optimal configurations according to our results. Furthermore, compared with prior machine learning approaches, our framework is very lightweight: it does not require offline training, hardware modification or OS coordination and it incurs negligible runtime overhead. Also, our machine learning methodology is very straightforward and can be easily generalized to solve other architecture problems.

8 CONCLUSION

This paper introduces *Memory Cocktail Therapy* (MCT), a general learning-based framework that manages to optimize combined techniques which utilize multiple dynamic trade-offs in NVM. With minimal performance overhead and no hardware modification, MCT manages to find the near-optimal configuration for the current application under a user-defined objective out of thousands of candidate configurations. MCT reduces the dimensionality of the configuration space with lasso regularization and selects three primary features: *fast_latency*, *slow_latency* and *write cancellation*. These are general features in NVM techniques so that our framework can also be applied to the optimization of other NVM techniques. Moreover, MCT manages to accommodate both fine-grained and coarse-grained phases by phase detection and cyclic-fine-grained runtime sampling.

We implement MCT using both gradient boosting and quadratic regression with lasso. We demonstrate that MCT using gradient boosting achieves better optimization results on average for all applications. For example, to guarantee an 8-year lifetime, achieve an IPC that is within 95% of the maximum, and minimize energy, MCT using gradient boosting improves the performance by 9.24% and reduces energy by 7.95% compared to the best static configuration. Compared with the ideal configuration in each application, MCT achieves 94.49% of the performance with only 5.3% more energy consumption (geometric mean).

9 ACKNOWLEDGEMENTS

The effort on this project is partially funded by the U.S. Government under the DARPA BRASS program, by the Dept. of Energy under DOE DE-AC02-06CH11357, by the NSF under CCF 1439156, and by a DOE Early Career Award.

REFERENCES

[1] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. 2015. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*. IEEE, 725–737.

[2] 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. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.

[3] Ramazan Bitirgen, Engin Ipek, and Jose F Martinez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 318–329.

[4] Zhaoxia Deng, Ariel Feldman, Stuart A Kurtz, and Frederic T. Chong. 2017. Lemonade from Lemons: Harnessing Device Wearout to Create Limited-Use Security Architectures. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 361–374.

[5] Zhaoxia Deng, Lunkai Zhang, Diana Franklin, and Frederic T Chong. 2015. Herniated hash tables: Exploiting multi-level phase change memory for in-place data expansion. In *Proceedings of the 2015 International Symposium on Memory Systems*. ACM, 247–257.

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

[7] Christophe Dubach, Timothy M Jones, Edwin V Bonilla, and Michael FP O'Boyle. 2010. A predictive model for dynamic microarchitectural adaptivity control. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 485–496.

[8] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. 2003. Characterizing and predicting program behavior and its variability. In *Parallel Architectures*

and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on. IEEE, 220–231.

[9] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. 2010. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ACM Sigplan Notices*, Vol. 45. ACM, 335–346.

[10] Jerome H Friedman. 2002. Stochastic gradient boosting. *Computational Statistics & Data Analysis* 38, 4 (2002), 367–378.

[11] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. 2004. Method-level phase behavior in Java workloads. *ACM SIGPLAN Notices* 39, 10 (2004), 270–287.

[12] Dibakar Gope and Mikko H Lipasti. 2014. Bias-free branch predictor. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 521–532.

[13] Edward Grochowski and Robert E Fontana Jr. 2012. Future technology challenges for NAND flash and HDD products. *Flash Memory Summit* (2012).

[14] Andrew Hay, Karin Strauss, Timothy Sherwood, Gabriel H Loh, and Doug Burger. 2011. Preventing PCM banks from seizing too much power. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 186–195.

[15] Henry Hoffmann. 2015. JouleGuard: energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 198–214.

[16] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 39–50.

[17] Aamer Jaleel. 2006. SPEC CPU 2006 Memory workload characterization. http://www.jaleels.org/ajaleel/. (2006).

[18] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. 2014. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling.. In *FAST*. 61–74.

[19] Daniel A Jiménez and Calvin Lin. 2001. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE, 197–206.

[20] Benjamin C Lee and David M Brooks. 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ACM SIGOPS Operating Systems Review*, Vol. 40. ACM, 185–194.

[21] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 2–13.

[22] Hsien-Hsin S Lee, Gary S Tyson, and Matthew K Farrens. 2000. Eager writeback-a technique for improving bandwidth utilization. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. ACM, 11–21.

[23] Bing Li, ShuChang Shan, Yu Hu, and Xiaowei Li. 2014. Partial-SET: write speedup of PCM main memory. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 1–4.

[24] Qingan Li, Lei Jiang, Youtao Zhang, Yanxiang He, and Chun Jason Xue. 2013. Compiler directed write-mode selection for high performance low power volatile PCM. *ACM SIGPLAN Notices* 48, 5 (2013), 101–110.

[25] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 469–480.

[26] Owen Y Loh and Horacio D Espinosa. 2012. Nanoelectromechanical contact switches. *Nature nanotechnology* 7, 5 (2012), 283–295.

[27] Joao Mendes-Moreira, Carlos Soares, Alípio Mário Jorge, and Jorge Freire De Sousa. 2012. Ensemble approaches for regression: A survey. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 10.

[28] Nikita Mishra, John D. Lafferty, and Henry Hoffmann. 2017. ESP: A Machine Learning Approach to Estimating Application Interference. To appear in proceedings of the 14th IEEE-International conference on Automatic Computing, 2017. A preprint available at: http://people.cs.uchicago.edu/~hankhoffmann/mishra-icac2017.pdf. (2017).

[29] Nikita Mishra, Huazhe Zhang, John D Lafferty, and Henry Hoffmann. 2015. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 267–281.

[30] Prashant J Nair, Chiachen Chou, Bipin Rajendran, and Moinuddin K Qureshi. 2015. Reducing read latency of phase change memory via early read and turbo read. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 309–319.

[31] M. Poremba and Yuan Xie. 2012. NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories. In *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*. 392–397.

[32] Moinuddin K Qureshi, Michele M Franceschini, Ashish Jagmohan, and Luis A Lastras. 2012. PreSET: improving performance of phase change memories by exploiting asymmetry in write times. *ACM SIGARCH Computer Architecture News* 40, 3 (2012), 380–391.

[33] Moinuddin K Qureshi, Michele M Franceschini, and Luis A Lastras-Montano. 2010. Improving read performance of phase change memories via write cancellation and write pausing. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 1–11.

[34] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 381–391.

[35] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 14–23.

[36] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 24–33.

[37] Hebatallah Saadeldeen, Diana Franklin, Guoping Long, Charlotte Hill, Aisha Browne, Dmitri Strukov, Timothy Sherwood, and Frederic T Chong. 2013. Memristors for neural branch prediction: a case study in strict latency and write endurance challenges. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 26.

[38] SanDisk. 2015. SanDisk and HP launch partnership to create memory-driven computing solutions. (2015). https://www.sandisk.com/about/media-center/press-releases/2015/sandisk-and-hp-launch-partnership.

[39] Robert E Schapire. 1990. The strength of weak learnability. *Machine learning* 5, 2 (1990), 197–227.

[40] Stuart Schechter, Gabriel H Loh, Karin Strauss, and Doug Burger. 2010. Use ECP, not ECC, for hard failures in resistive memories. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 141–152.

[41] Xipeng Shen, Yutao Zhong, and Chen Ding. 2004. Locality phase prediction. *ACM SIGPLAN Notices* 39, 11 (2004), 165–176.

[42] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *ACM SIGARCH Computer Architecture News*, Vol. 30. ACM, 45–57.

[43] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase tracking and prediction. In *ACM SIGARCH Computer Architecture News*, Vol. 31. ACM, 336–349.

[44] StatTrek.com. 2017. Coefficient of determination. http://stattrek.com/statistics/dictionary.aspx?definition=coefficient_of_determination. (2017).

[45] Guangyu Sun, Xiangyu Dong, Yuan Xie, Jian Li, and Yiran Chen. 2009. A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*. IEEE, 239–249.

[46] Elvira Teran, Zhe Wang, and Daniel A Jiménez. 2016. Perceptron learning for reuse prediction. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[47] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), 267–288.

[48] Rujia Wang, Youtao Zhang, and Jun Yang. 2016. ReadDuo: Constructing Reliable MLC Phase Change Memory through Fast and Robust Readout. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE, 203–214.

[49] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 476–488.

[50] Cong Xu, Dimin Niu, N. Muralimanohar, R. Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. 476–488.

[51] Lunkai Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T Chong. 2016. Mellow writes: Extending lifetime in resistive memories through selective slow write backs. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 519–531.

[52] Lunkai Zhang, Dmitri Strukov, Hebatallah Saadeldeen, Dongrui Fan, Mingzhe Zhang, and Diana Franklin. 2014. SpongeDirectory: Flexible sparse directories utilizing multi-level memristors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 61–74.

[53] Mingzhe Zhang, Lunkai Zhang, Lei Jiang, Zhiyong Liu, and Frederic T Chong. 2017. Balancing Performance and Lifetime of MLC PCM by Using a Region Retention Monitor. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 385–396.

[54] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *ACM SIGARCH computer architecture news*, Vol. 37. ACM, 14–23.

[55] Yanqi Zhou, Henry Hoffmann, and David Wentzlaff. 2016. CASH: Supporting IaaS customers with a sub-core configurable architecture. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 682–694.