J Supercomput (2018) 74:3875–3903
https://doi.org/10.1007/s11227-018-2438-y

NVM Streaker: a fast and reconfigurable performance simulator for non-volatile memory-based memory architecture

Danqi Hu[1,2] · Fang Lv[1] · Chenxi Wang[1,2] · Hui-Min Cui[1,2] · Lei Wang[1] · Ying Liu[1] · Xiao-Bing Feng[1,2]

Published online: 2 June 2018

Abstract The high density, low power consumption non-volatile memory (NVM) provides a promising DRAM alternative for the in-memory big-data processing applications, e.g., Spark, It is significant to simulate the behaviors when NVMs are deployed into the area of big-data processing before their widespread use in market. However, existing simulation approaches are not applicable for big-data processing due to two reasons. First, some approaches require complicated hardware and/or OS supports. Second, cycle-level or function-level simulations are too time-consuming to simulate the whole software stack of big-data processing. Therefore, the complexity and expensive time cost in NVM simulation have dramatically dragged down the integrated research of big data with NVM. This paper proposes a fast and reconfigurable simulation method, called NVM Streaker, which does not need complex hardware or OS supports. It simulates NVM access costs using disturbed DRAM accesses and commonly configurable hardware parameters. It is fast since we use DRAM accesses and change its access costs to simulate NVM access costs, thus enabling to simulate the whole software stack to run Spark applications. It is reconfigurable since we enable users to configure the disturbed memory access costs, in order to simulate dif-

This research is supported by National Key R&D Program of China under Grants Nos. 2016YFB1000402, 2016YFB1000200 and 2016YFB0200504; the NSFC under Grants Nos. 61521092, 61672492, 61303053, 61432016 and 61402445.

✉ Fang Lv
flv@ict.ac.cn

1 State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

2 University of Chinese Academy of Sciences, No. 19(A) Yuquan Road, Shijingshan District, Beijing, China

ferent NVM access costs. The experimental results show that we can simulate Spark applications with almost negligible cost and high efficiency.

Keywords NVM · Reconfigurable · Fast simulation · Big data

1 Introduction

The proliferation of big-data applications has raised new challenges for data storage and processing technologies. The characteristics of high density and low power make NVM a possible alternative of DRAM to cater for the increasing demands in in-memory computation of big data. Therefore, the integrated research of big data and NVM is becoming a promising topic [6, 33, 37, 43].

However, the existing NVM techniques cannot meet the needs of big-data researches directly. Although studies on NVM are at the forefront, the commercial unavailability of NVM-based memory architecture has dragged down the combination of NVM and big-data researches. Moreover, the durability of NVM, e.g., the write life problem [10, 16, 19, 31], makes it hard to sustain high pressure of performance experiments which further restricts the researches. Therefore, a more helpful simulation technique for NVM architecture is in pressing need.

The state-of-art researches need better NVM platform simulation methods. Explorations on persist memory system [34] have thorough considerations for NVM-specific characteristics. User applications can utilize these techniques with customized interfaces for NVM, which impose higher requirements on developers. Architectural-level NVM simulators [8, 11, 18, 24–26, 28] for physical designs concerned more on hardware details such as memory cells, transistor size. Some works either relied on complicated hardware and OS supports [5, 24], or were very time-consuming. Some simulators or emulators only provided calculation for memory latency or bandwidth with hardware details without supports for real runtime simulation at all [8, 26, 30]. Therefore, a new simulation method for NVM-based memory architecture is necessary to facilitate performance-related big-data researches.

Performance-oriented software researches [39, 40] on NVM always focus on software exploitable key characteristics such as longer access costs. In fact, higher memory costs can be simulated by making good use of the existing supports from mainstream NUMA servers [23]. NUMA server has distributed memory regions. The remote memory region has longer access costs than the native memory region which make it adaptive to work as the simulated NVM region naturally. NUMA server has many tunable parameters which can generate even more simulated memory access costs. Furthermore, the memory access costs can be prolonged by means of software-auxiliaried contentions on the shared memory bandwidth. The contentions are confined inside the simulated NVM region in order to limit their overheads. All in all, we can realize fast NVM simulation with the aid of software methods on NUMA servers at acceptable simulation costs.

In this paper, we propose a fast and reconfigurable simulation method, NVM Streaker. It works efficiently with measures mentioned in the previous paragraph while without any complicated modifications on the hardware, OS and user applications.

NVM Streaker simulates the longer memory access costs for NVM in two steps. First it simulates a number of longer memory access costs through configuring the tunable hardware parameters on NUMA servers. Then, it obtains more simulated values under the auxiliary of a bandwidth scrambling task named the **disturber**. The disturber keeps issuing memory demands into the shared memory bandwidth which conflict with the memory operations from the user application. The resulting memory conflicts prolong the DRAM access costs and lead to higher simulated NVM access costs finally. Thus, the software disturber can produce a much wider simulation scope for NVM. For example, on our experimental server, the parameter reconfiguration can produce simulated NVM access costs in $2.5x$–$3.2x$ compared to the native DRAM costs. The software disturber can widen the range to more than $5.1x$. These two measures make NVM Streaker able to work for performance simulation with low complications and at acceptable costs.

NVM Streaker devotes to efficient performance simulation for integrated researches with NVM. Besides the low costs and the low complexity, both the parameter configuration and the software disturber endow it a high configurability. User can configure different NVM environments through adjusting the tunable parameters either in the server hardware or in the software disturber. In addition, we propose Fast Calculation for much easier simulation, which needs no real simulation executions at all. Both the carefully designed NVM Streaker, and Fast Calculation can support simulations with efficiency and accuracy.

The contributions of our work are as follows:

- We propose an easily implemented simulation method, NVM Streaker, for NVM-based memory architectures. It supports key characteristics of NVM including the higher memory access costs, and the asymmetric costs between read and write requests while without any sophisticated modifications on hardware, OS and user applications. NVM Streaker conducts fast simulation by means of hardware parameter reconfiguration and a software-auxiliaried disturbing method. NVM Streaker enables users to configure different NVM environments through adjusting the tunable parameters either in the server hardware or in the software disturber.
- We propose an even faster method based on NVM Streaker which is named Fast Calculation. It can simulate NVM environment accurately and more conveniently via calculation, while saving the simulated execution process (without execution).
- NVM Streaker is able to work for performance simulation or analysis with high accuracy. Experiments with Spark benchmarks verify NVM Streaker's low time costs (almost negligible), the high accuracy, and convenience in reconfiguration in the meantime.

The rest of the paper is organized as follows: Sect. 2 presents the implementation of our simulator, NVM Streaker. Section 3 details the key design of necessary parameters for the simulator. Section 4 details the key design for the disturbing tasks used in the simulator. Experiments and evaluations are discussed in Sect. 5. Section 6 discusses the related work. Section 7 draws the conclusion and discusses the future work.

2 NVM Streaker: a fast and reconfigurable simulation method

NVM Streaker is a performance simulator for NVM architecture. It supports key characteristics of NVMs, including its higher memory access cost and asymmetric memory cost between read and write operations. It aims at efficient simulation at acceptable time cost, with high configurability, and low complexity.

In order to achieve the above objectives, the key point of NVM Streaker is to limit its simulation on memory operations while without influences on other unrelated operations or function units. This idea can lead to low complexity and low time costs directly.

NVM Streaker is designed as a cooperated method between a parameter layer and a software disturber. These parts jointly contribute to the simulated NVM performance for user applications. First, the parameter layer produces several simulated NVM access costs via tuning the adjustable hardware parameters of the server. The server configured with different hardware parameters works as different NVM environments. User applications can run under these simulated environments directly. However, these discrete values are insufficient for thorough performance simulation and analysis. Hence, a software disturber is used to further delay the memory access costs for more NVM environments. The disturber keeps competing for the shared memory bandwidth with the user application by launching memory operations constantly. Under this way, all target memory operations from user applications are interfered, prolonged, and evolve into the simulated NVM access costs finally. With these design details, NVM Streaker can efficiently limit its simulation costs and complexity.

2.1 The framework of NVM Streaker

Figure 1 illustrates the framework of NVM Streaker. It is a two-layer design which includes a parameter layer and a software disturber. User applications can achieve their desired NVM simulation performance under the joint effects of these two parts. NVM Streaker can be built on any mainstream NUMA servers. On such kind of memory architecture, the memory spaces are classified into the native region and the remote region naturally. For a user application which is launched on CPU0, the remote region always has relatively higher memory access costs than the native region. Therefore, in our work, we use the remote region as **the simulated NVM region**. All simulated access costs for NVM are measured inside this region.

The parameter layer simulates NVM via parameter configurations on the hardware. Most of the servers provide some adjustable parameters which include **the CPU frequency**, **the memory frequency** and **the memory channel number**. Variations in these parameters lead to different simulated memory access costs at much lower simulation time costs than architecture-level simulators.

The software disturber further widens the simulated scope for NVM access costs by means of the memory disturbing mechanism. As shown in Fig. 2, the memory operations to be simulated from user applications which are launched on CPU0 are placed in the simulated NVM region. The software disturber on CPU1 keeps launching mem-

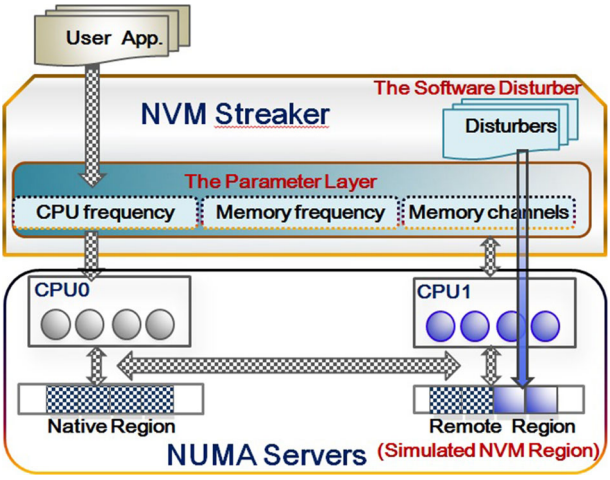

Fig. 1 Framework of NVM Streaker

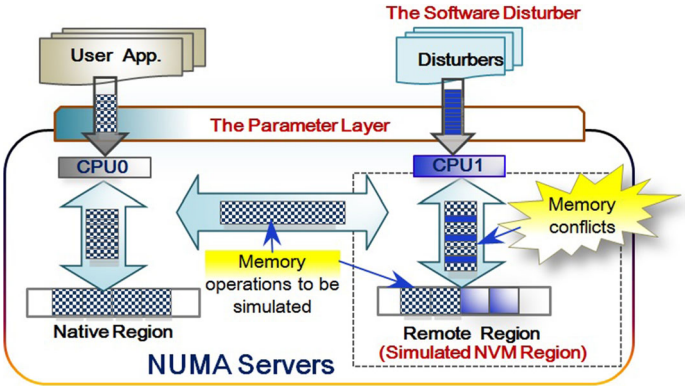

Fig. 2 The software disturber of NVM Streaker

ory operations constantly into this simulated NVM region either. Both the disturber and the user applications compete for the shared memory bandwidth as the annotation box refers to. The resulting memory conflicts lead to prolonged memory access costs for all memory operations, which are treated as the simulated NVM access costs for user applications. Through tuning the memory bandwidth of the software disturber, different NVM access costs can be produced.

From this mechanism, we can see that NVM Streaker's fast simulation is ensured by the jointly cooperated layers. This advantage can be clarified by the above designs:

- NVM Streaker works via direct running on real servers, neither on cycle-level simulators or function-level simulators, nor on FPGA simulators. These characters make it easy to apply and more time-saving.
- The parameter layer works through the parameter reconfiguration which has almost negligible simulation costs.

- The software disturber limits its influences on the simulated memory operations, while without interferences on other type of operations. Variations on the performance can be contributed to the simulated memory operations. This makes it more efficient for any further performance analysis.

Another advantage of NVM Streaker is that it is highly reconfigurable. It can configure the simulated environment flexibly via the hardware parameter configuration. Moreover, more simulated environments can be set up through tuning the disturbance degree of the software disturber.

2.2 The evaluation matrix with NVM Streaker

Different to architecture-level simulators or other hardware simulators, it is difficult for NVM Streaker to differentiate read and write operations. It cannot produce explicit latency for read and write operation. It is also not our desired way to insert delays precisely after every memory operations since it is time-consuming. In our work, we use a synthesized performance index, the **average memory access cost** (**AMAC**), to quantify and depict the simulated memory access costs for NVM Streaker. This index depicts the average access costs for both read and write operations. It is the total main-memory access number divides the total time cost to finish those accesses under the simulated NVM environment. More design details in our later work illuminate that it is sensitive to the diversified memory characteristics of applications and it is qualified for performance simulation for performance-oriented software researches.

AMAC can work for performance simulation and evaluation. AMAC is calculated directly, which is dividing the total access time cost by the total main-memory access number. On the contrary, for applications with implicit quantity of memory operations, we analyze the application via the simulated performance, i.e., the whole runtime in simulation.

NVM Streaker is helpful for performance-oriented software researches. It can facilitate the performance analysis via variations on AMAC or on performance. Through contrasting the variations with and without optimizations, the researchers can obtain information about their innovated techniques.

3 Detailed design of the parameter layer

Parameter configuration on the hardware is always an ideal simulation method for performance-oriented researches. They can complete the simulation process at very low time costs. NVM Streaker is built on mainstream servers with NUMA supports. Naturally, the distributed memory spaces can work as DRAM regions and NVM regions, respectively. Moreover, mainstream architectures always provide certain tunable parameters. Among them, **the CPU frequency**, **the memory frequency** and **the memory channel number** are three of the tunable parameters which relate closely to the memory access cost. The parameter layer can produce a number of memory access costs through tuning these parameters.

Although the CPU frequency influences the whole applications, for memory-intensive applications which we focus on, this parameter is useful to produce much higher memory access costs for NVM simulation. The two parameters of the memory frequency and the memory channel can affect the memory costs directly without side effects on other types of operations. These three parameters cooperate to influence all memory operations.

The parameter layer implements a fast performance simulation. The way it works can influence the execution time of the user application directly while without lengthy expenses as from traditional cycle-level simulators, trace-level or function-level simulators. Moreover, these adjustable parameters make NVM Streaker reconfigurable for NVM environments with different access costs.

3.1 Environmental description

To be clarity, we define the hardware working circumstance with the above three adjustable parameters as $\langle \mathrm{CPU}, \mathrm{MEM}, \mathrm{CH} \rangle$. **CPU** and **MEM** stand for the working frequencies of the architecture. **CH** stands for the memory channel number in the working machine. We use $\mathrm{DRAM}^{\langle \mathrm{CPU,MEM,CH} \rangle}$ and $\mathrm{NVM}^{\langle \mathrm{CPU,MEM,CH} \rangle}$ to distinct the working environment between the DRAM region and the simulated NVM region, respectively. Thus, the working circumstance is determined with vector $[\mathrm{DRAM}^{\langle \mathrm{CPU,MEM,CH} \rangle}, \mathrm{NVM}^{\langle \mathrm{CPU,MEM,CH} \rangle}]$. More details about our platform can be found in Sect. 5.

3.2 Limitation of the parameter layer

The major inefficiency of the parameter layer is that it has very narrow simulation scope. The parameter layer is constructed on a few discrete hardware circumstance parameters, which lead to few simulated NVM costs. These finite values are not qualified for thorough performance analysis. Thus, the configurability is very limited. Hence, a better method with much wider simulation scope is necessary for the big-data researches on NVM-based memory architectures.

4 Detailed design of the software disturber

The software disturber plays a key role for the NVM simulation. It adopts a disturbing mechanism to enhance the simulation capability for NVM Streaker. Under this mechanism, an application co-runs with a series of disturbers. The obtained memory access costs of the application are treated as the average memory access costs (the **AMAC** introduced in Sect. 2.2) for NVM. Different combinations between the application and the disturber lead to different AMAC. This mechanism illuminates that the simulated AMAC values for an application cannot be determined by any single party. They are determined by both the co-running disturber and the co-running application concurrently.

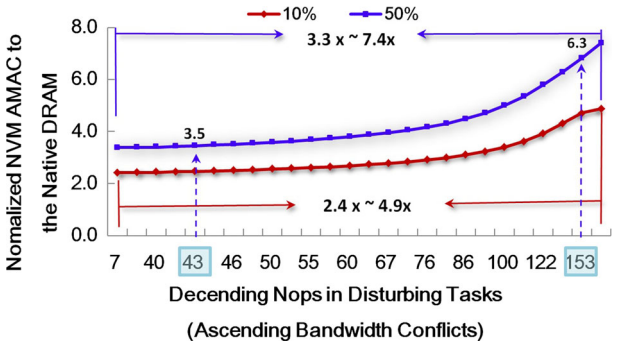

Fig. 3 Illustration for the common-decision principle of NVM Streaker. Different co-runner combinations lead to different simulated AMAC values

Figure 3 illustrates the common-decision principle of the disturbing mechanism, i.e., different co-runner combinations can lead to different simulated AMAC for NVM. In the figure, two tasks with different memory properties produce two groups of memory access costs which form the two curves when co-running with the same group of disturber tasks. The two sample applications have the same LLC miss rate, while different read–write miss rates, 10% write miss rate (in red) and 50% write miss rate (in blue), respectively. The task in the blue curve can simulate in 3.3*x*–7.4*x* AMAC normalized to the native DRAM cost, while the task in the read curve can simulate in a narrower scope of 2.4*x*–4.9*x*. Variations in the co-running combinations for the same task can lead to different simulated AMAC for NVM. For the task with 50% write rate, it can obtain 3.5*x* AMAC through co-running with a disturbing task with memory bandwidth of 43 M/s. It can obtain 6.3*x* AMAC that of DRAM through co-running with a disturbing task with memory bandwidth of 153 M/s. Therefore, for a given application, NVM Streaker can obtain unique AMAC or performance through co-running it with a disturber with specified memory property.

In order to quantify and depict all simulated AMAC values for NVM, we need a series of **disturbers** to generate different degrees of disturbing memory operations which aid for various memory access costs. Besides this kind of tasks, we need a series of tasks as representative samples, named the **imitator**, to imitate user applications' memory properties. We collect all possible simulated AMAC values through co-running each of these sample tasks with all disturbers. All these AMAC values can be treated as the possible simulated NVM access costs with NVM Streaker. For any new application, we can set up the simulated environment in accordance with its most similar imitator task. Thus, the new application can obtain designated NVM environments through proper combinations with disturbers. This is the ***basic idea*** of the software disturber.

The mechanism of NVM Streaker determines that it is very important to ensure of the accuracy of simulator. The imitators should be as analogous as possible to represent real applications. Thus, a user application can obtain consistent simulation results with its most similar imitator. This means that through observation the gap between the simulated AMAC values from the user application and its most similar imitator, we can make a judge on the accuracy of NVM Streaker.

4.1 Kernel designs in the software disturber

In order to obtain the disturber set and the imitator set, a training process is adopted in our work. The training process helps to establish both the disturbers and the imitators while eliminating those who act similarly. Finally, we can obtain a set of disturbers which produce competing memory demands evenly at different speeds. And we can obtain a set of imitators which present high performance sensitivity to different disturbers.

4.1.1 The kernel design for the disturber and the imitator

NVM has some recognized key performance-related characteristics which are covered in most of the researches, such as longer access latency especially slow writes [34], and the asymmetric costs between read and write request. We support both of these two features in our kernel design.

Real-user applications are diversified. The differences in the thread number, the memory bandwidth, and the read–write miss ratio, etc., all lead to diversified memory behaviors. In order to support the diversified memory behavior for real applications, we take the above key factors in the kernel design for both of the disturber and the imitator. The kernel includes adjustable memory properties. Through tuning the miss rate of the last level cache (LLC), asymmetric read–write miss rate, and thread number, tasks with different memory behaviors can be generated for training.

The kernel of the disturber Figure 4 demonstrates the disturber's kernel. The kernel's design is a commonly used benchmark for memory bandwidth, which is similar to the bubble's design [21, 44]. It adopts three important features for fine-grained bandwidth tuning: (1) regular memory accesses specified by Line 13; (2) configurable memory bandwidth specified by variables of **NOP** and **STRIDE**; (3) uniformly continuous memory behavior. It is worth to mention that the main mission of the disturbing task is to generate memory conflicts, which are either from read operations or write operations. For simplicity, we limit the operation type to read only. Besides, all disturbers are designed with full-loaded threads (6 threads on our 6-core/CPU server).

All disturbers are defined within a set $\boldsymbol{Dis}$**_Set** : $\{D_1, \ldots, D_p\}$. Each D_i specifies the memory bandwidth at which this disturber keeps launching evenly. As the kernel's memory demands dial up from D_1 to D_p, the memory bandwidth demands increase monotonically. In the process of dialing up from D_1 to D_p ascendingly, we measure the AMAC of the co-running imitator task. The acquired AMAC values stand for the simulated memory costs for different NVM environments.

The kernel of the imitator Due to the diversified properties as mentioned above, user applications always behave differently and present different performance during co-running with different disturbers. NVM Streaker uses imitators to imitate different user applications. Key properties including the LLC miss rate, the read–write miss rate, and the thread number are included in the design of imitator tasks.

Figure 5 demonstrates the imitator's kernel design. It uses the variable **NUM_THREADS** to control the number of concurrent threads in the imitator or

Fig. 4 The kernel of the disturber

Algorithm Disturber-Kernel: The kernel produces memory demands at tunable speed constantly. These demands compete for the shared memory bandwidth, thus interfere with the co-runner's memory accesses.
Input: NULL
Output: NULL

```
1:  #define SLICE    (32*1024*1024)
2:  #define STRIDE   8
3:  #define NOP      100
4:  uint64_t *a;
5:  void* count(void* arg)
6:  {
7:    size_t id = *((size_t*)arg);
8:    uint64_t* start_sec= a+SLICE*id;
9:    uint64_t* end_sec=
10:             start_sec + SLICE;
11:   for(;;){
12:     for(i=start_sec;i<end_sec;){
13:       *(end_sec) += *(i+=2*STRIDE);
14:       for(m=0;m<NOP;m++)
15:         asm("nop");
16:     }
17:   }
18: }
```

the user task. It uses several memory-related variables to adjust the memory behavior. These variables can make them flexibly configured for a variety of tasks with various memory behaviors.

In the figure, the variable **MEM_PER_ITER** specifies memory operations in every iteration. Almost every memory operation in the kernel can lead to a cache miss. The variable **NOP** and **STRIDE** can control the memory bandwidth. They can switch the training kernel from memory-intensive type to less memory-intensive one. The read–write miss rate can be tuned via the variable of **WR**. All imitators are designed with a set **Eval_Set**:$\{\mathrm{WR}_0, \mathrm{WR}_1, \ldots, \mathrm{WR}_q\}$, while WR_j stands for the proportion of write misses of a kernel, which is between 0 and 100%. For example, WR_j of 25% represents that write misses occupy 25% of all cache misses. The imitators in the set have ascending write miss ratios.

With the above variables, we generate three kinds of imitators which have different memory behaviors for our investigation:

- Memory-intensive imitator: It represents memory-intensive applications which are full of LLC misses. They are developed from a memory-intensive imitator kernel with maximum cache miss rate, i.e., almost every memory access encounters a cache miss.
- Less memory-intensive imitator: It represents less memory-intensive applications which have relatively less LLC misses. They are created through tuning the variable of **NOP** and STRIDE in the imitator kernel.
- Imitator with variable number of threads: It represents applications with variable thread number.

Fig. 5 The kernel of the imitator

Algorithm Imitator-Kernel : It imitates the memory behaviors of user applications with tunable parameters on the memory bandwidth, the read-write LLC miss ratio and the thread number .

Input: NULL

Output: memory transfer rates

```
1:  #define N                  80000000
2:  #define NTIMES             200
3:  #define STRIDE               8
4:  #define MEM_PER_ITER 100
5:  #define NOP                100
6:  #define NUM_THREADS  6
7:
8:  for (k=0; k<NTIMES; k++){
9:     m = N-MEM_PER_ITER*STRIDE;
10: #pragma omp parallel for
11:   for(j=0;j<m;
12:       j+=MEM_PER_ITER*STRIDE){
13:     sum+=a[j];
14:     sum+=a[j+STRIDE];
15:     ...
16:     sum+=a[j+(WR-1)*STRIDE];
17:     a[j+WR*STRIDE]=j+1;
18:     a[j+(WR+1)*STRIDE]=j+2;
19:     ...
20:     a[j+(MEM_PER_ITER-1)*STRIDE]=
21:           j+MEM_PER_ITER-WR;
22:     for(i=0;i<NOP;i++)
23:           asm("nop");
24:   }
  }
```

In the following subsections, we will study the impacts on NVM AMAC in detail with these three kinds of imitators.

4.1.2 The training process

The training process is to (1) establish a set of disturbers which produces various degrees of interference and leads to different simulated memory access costs for user applications, (2) establish a set of imitators which has different memory properties and present distinct memory access costs under different memory contentions, and (3) guild the user application to construct their designated simulation environment.

In the training process, each disturber D_i in **Dis_Set** can produce a certain degree of interference. Each imitator WR_j can present different AMAC values when co-running with a disturber D_i. All AMAC values form a performance curve for each imitator after co-running with all disturbers. Thus, the training process can produce $p \times q$ groups of latency curves for all the disturbers and the imitators. With all these curves, we carefully eliminate some unqualified tasks which have similar performance with others, and select out n disturbers to form the final **Dis_Set** : $\{D_1, \ldots, D_n\}$ and m imitators to form the final **Eval_Set** : $\{\mathrm{WR}_1, \ldots, \mathrm{WR}_\mathrm{m}\}$. The tasks in these two sets can show distinct AMAC (performance) during the training process.

Training with memory-intensive imitators Figure 6 demonstrates the training process for the disturbers with memory-intensive imitators. During the training process, all memory conflicts are triggered inside the simulated NVM region. Experiments from (a)–(f) are collected under 6 different hardware parameter configurations. In each figure, 11 memory-intensive imitators with write miss ratios ranging from 0 to 100% are adopted. They are developed from the memory-intensive imitator kernel with maximum cache miss rate. More than 20 disturbers are involved in the evaluation process. *X*-axis stands for the variable **NOP** which is used to control the disturbing task's memory bandwidth demands. The descending of **NOP** value can produce ascending memory bandwidth. *Y*-axis stands for the normalized simulated NVM AMAC to the native DRAM. Consider Fig. 6a for example, under the configuration of [DRAM$^{\langle 2.67,1.33,3\rangle}$, NVM$^{\langle 2.67,1.33,1\rangle}$, the hardware parameter layer can produce simulated NVM access costs ranging from 2.5x–3.2x. The software disturber with 240 nops per iteration can further widen the range up to 5.1x as the arrow denotes. However, it should be pointed out that in order to obtain stable data, those disturbers and imitators which fluctuate severely will not be adopted and are eliminated from the training process.

From these figures, we can observe that experiments under different hardware parameters have a similar trend: an ideal, stable and monotone AMAC curve for each imitator. This means that for every imitator, the disturbers can produce simulated memory access costs ascendingly when ranging the variable of **NOP** from 900 to 60. This is a common phenomenon which applies to all the 6 hardware configurations.

Training with less memory-intensive imitators We measure AMAC from another point of view with less memory-intensive imitators. For clarity, we choose three kinds of kernels with WR of 0, 50% and 100%, respectively. We develop several less memory-intensive tasks from these three kernels through tuning the variable of **NOP** in the kernel. Thus, a series of imitators with ascending memory bandwidth demands can be created.

Figure 7 demonstrates that experiments with less memory-intensive tasks still can follow the similar rule with the memory-intensive tasks. A stable and monotone AMAC curve for each imitator can be formed. In each figure, 11 imitators with descending LLC misses per cycle (17,902–52 LLC misses per 106 cycles) are adopted. These tasks are created with ascending **NOP** which ranges from 0 to 300. *X*-axis stands for the same disturbers as in Fig. 6. *Y*-axis stands for the simulated AMAC normalized to that on native DRAM for 11 imitators. Experiments show that, similar to the memory-intensive reports, less memory-intensive imitators also can react differently to the same group of disturbers. We can observe an ideal, stable and monotone AMAC curve for each imitator when the variable of **NOP** in the disturbers ranges from 900 to 60.

Training with imitators of variable thread number We take the thread number into the design of the imitator. For an application, the variation in the thread number can exert distinct influences on the AMAC values.

Figure 8 demonstrates the experiments with thread numbers of 1-thread, 2-thread and 6-thread, respectively. Experiments from (a)–(c) are conducted with three kinds imitators. Each figure includes imitators extended from WR of 0, 50 and 100%, respec-

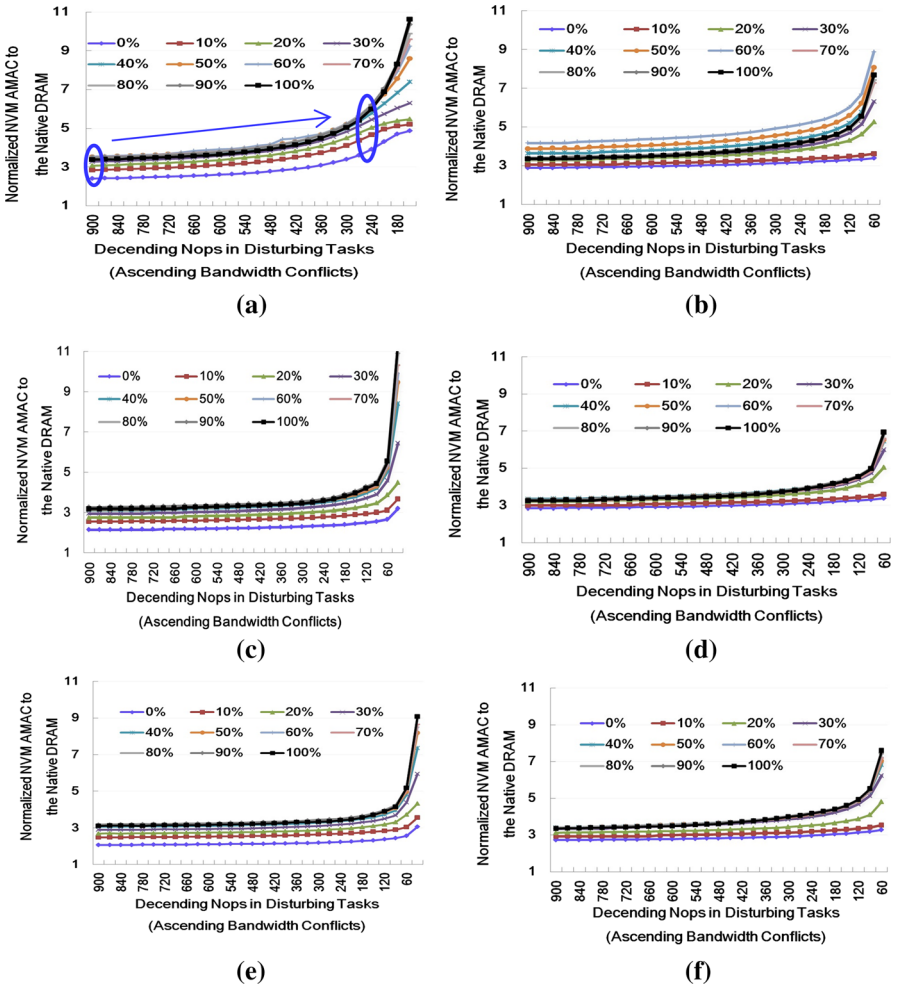

Fig. 6 Training process with memory-intensive evaluation tasks under 6 hardware parameter configurations. **a** Training process under [**DRAM**$^{\langle 2.67,1.33,3\rangle}$,**NVM**$^{\langle 2.67,1.33,1\rangle}$], **b** training process under [**DRAM**$^{\langle 2.67,0.8,3\rangle}$, **NVM**$^{\langle 2.67,0.8,1\rangle}$], **c** training process under [**DRAM**$^{\langle 2.13,1.33,3\rangle}$, **NVM**$^{\langle 2.13,1.33,1\rangle}$], **d** training process under [**DRAM**$^{\langle 2.13,0.8,3\rangle}$, **NVM**$^{\langle 2.13,0.8,1\rangle}$], **e** training process under [**DRAM**$^{\langle 1.6,1.33,3\rangle}$, **NVM**$^{\langle 1.6,1.33,1\rangle}$], **f** training process under [**DRAM**$^{\langle 1.6,0.8,3\rangle}$, **NVM**$^{\langle 1.6,0.8,1\rangle}$]

tively. The relation between thread number and the AMAC can be observed in these figures.

In each figure, x-axis stands for a group of disturbing tasks with ascending memory bandwidth demands. Y-axis stands for the AMAC of imitators normalized to that on native DRAM latency. From the figures, we can observe similar trends. The thread number does relate closely to the memory access costs of the imitators under the co-running mechanism. For the same imitator, the higher the thread number is, the higher the normalized value is. As the circle in Fig. 8a denotes, when co-running with the disturber of 900 **nops**, the normalized AMAC value of the 6-thread imitator

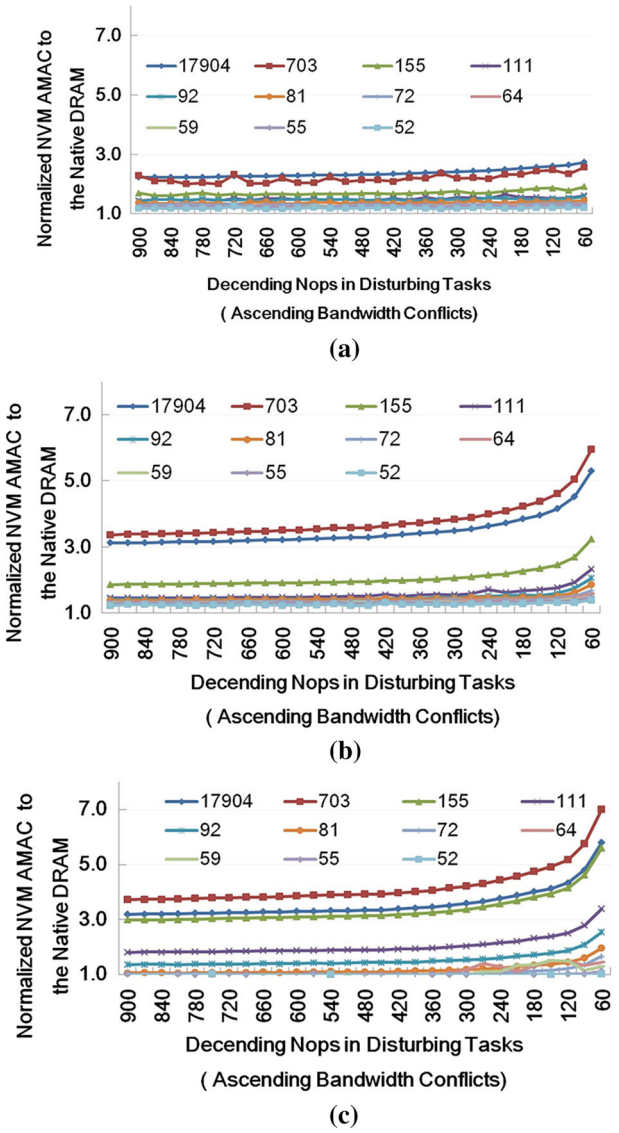

Fig. 7 Training process with less memory-intensive imitators. **a** Training process with imitators developed from WR = 0%, **b** training process with imitators developed from WR = 50%, **c** training process with imitators developed from WR = 100%

is 2.22. The normalized AMAC value of the 1-thread imitator is only 1.62. This is because that the higher the thread number is, the higher the memory bandwidth is. The increased demands from the imitator intensify the memory conflicts, which lead to higher AMAC.

Experiments with thread number show that the number of threads of the imitator will have its influences on the AMAC values. We still can observe an ideal, stable and

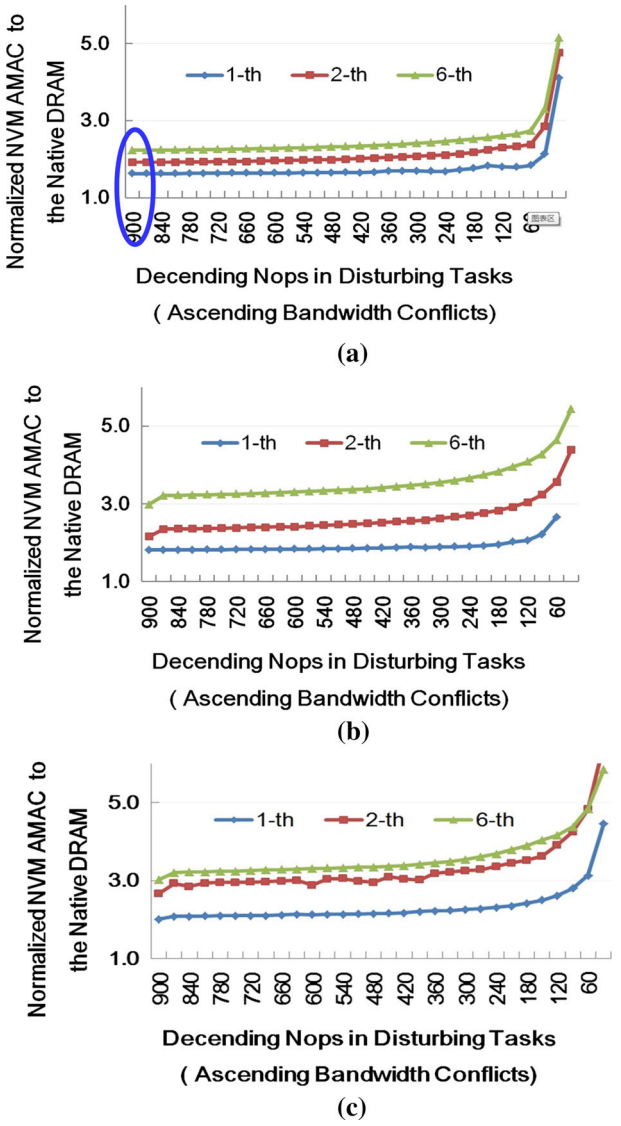

Fig. 8 Training process with variable thread number. **a** Training process with imitators developed from WR = 0%, **b** training process with imitators developed from WR = 50%, **c** training process with imitators developed from WR = 100%

monotone AMAC curve for each imitator when the variable of **NOP** in the disturbers ranges from 900 to 60.

Conclusion Common observations in the above experiments show that users can make good use of NVM Streaker in the following two aspects:

Table 1 Key factors in the training process

Factors	Co-runners	
Co-running tasks	Imitator task on CPU0	Disturber task on CPU1
Read miss rate of LLC	$RLLC_0$	$RLLC_1$
Write miss rate of LLC	$WLLC_0$	–
Thread number	TH_0	$TH_1 = 6$

On the one hand, the findings with both memory-intensive imitators and less memory-intensive imitators indicate how to build the simulation environment for user applications. We can create a designated NVM simulation environment for an imitator through proper combination between the disturber and this imitator task. For a given user application, we can pick out its most similar imitator task which have similar memory properties. Then we can establish designated NVM simulation environment through proper combination between this imitator and the disturbers. *This is the basic idea to guide simulation for real-user tasks with the software disturber*.

On the other hand, users can obtain useful information about their new optimizations on future NVM architectures. Through contrasts between data with and without their new optimizations under certain simulated environment, users can know whether the new techniques are beneficial.

4.2 Fast Calculation based on NVM Streaker

The software disturber can facilitate the NVM simulation for real applications. However, we still need to arrange the co-running execution between the user application and the disturbers for simulation. In this subsection, we propose an even faster method named **Fast Calculation**. This method can simulate via calculation directly without any execution process. This improves the efficiency of simulation greatly.

Fast Calculation makes a good use of all training results. All experiments results are gathered and form a relation between the imitators (or the user application) and the disturbers. The relation formula can be fitted out by some mathematical software, e.g., MATLAB [47] in our work. As shown in Table 1, there are 5 necessary parameters from the two co-runners which include the read miss rate of LLC, write miss rate of LLC, and thread number of the two co-runners. As for the disturbers, we use full-loaded memory-read intensive disturbers (which has 6 threads in our environment on the remote CPU). Thus, we reduce the factor of the disturbing task to $RLLC_1$ only. Equations (1)–(3) estimate the potential AMAC with the left 4 factors (except TH_1), $RLLC_0$, $WLLC_0$,TH_0,$RLLC_1$.

In order to configure a NVM environment, the user needs to specify its memory property. Here, ***User_Property*** is defined as the property of user task or the imitator as in Eq. (1). It is synthesized from the read miss rate of LLC ($RLLC_0$), the write miss rate of LLC ($WLLC_0$) and the thread number of the user application (TH_0). Constant

factors of a_0 and a_1 are used to indicate the difference between server machines. All these constants can be obtained in the process of formula fitting with MATLAB in our work. Different servers always have different constant factors.

$$\text{User\_Property} = a_0 \cdot \frac{\text{RLLC}_0}{\text{TH}_0} + a_1 \cdot \frac{\text{WLLC}_0}{\text{TH}_0} \tag{1}$$

Similar details of another co-runner, the co-running disturber, are also necessary. It is synthesized from the two co-runners' properties. With these features, ***Corun_Property*** is defined for the specified co-running circumstance as in Eq. (2). Constant factors of β_0, β_1, β_2 are adopted similarly to those factors in (1).

$$\text{Corun\_Property} = \left(\beta_0 \cdot \frac{\text{RLLC}_0}{\text{TH}_0} + \beta_1 \cdot \frac{\text{WLLC}_0}{\text{TH}_0} + \beta_2\right) \cdot \text{RLLC}_1 \tag{2}$$

After plenty of experiments, we can estimate the simulated AMAC value, $\text{AMAC}_{\text{estimated}}$, with Eq. (3). This equation is to estimate the AMAC value with the given 4 key features of both the two co-runners. Constant factor of δ_0 is used in this equation.

$$\text{AMAC}_{\text{estimated}} = \delta_0 + \text{User\_Property} + \text{Corun\_Property} \tag{3}$$

The runtime performance, $P_{\text{estimated}}$, for a user task is composed of two parts, the memory-related part and the non-memory-related part. We use $\text{ratio}_{\text{mem}}$ for the performance calculation which is the ratio of memory instructions to all instructions. $P_{\text{estimated}}$ can be estimated with the following equation:

$$P_{\text{estimated}} = \text{AMAC}_{\text{estimated}} * \text{ratio}_{\text{mem}} + (1 - \text{ratio}_{\text{mem}}) \tag{4}$$

Fast Calculation can estimate the simulated AMAC or performance for given user applications. Given a particular hardware setting and memory properties of a user application, the user can configure different simulated environments with various disturbers. Furthermore, it also can help to determine proper disturbers with definite requirements on AMAC. The user can calculate RLLC_1 to obtain a proper disturber with which to constitute a desired simulation environment.

Both NVM Streaker and Fast Calculation can work efficiently and accurately. In the following experiment section, we will illustrate the low time costs, low complexity and high exactness for simulation with NVM Streaker.

5 Experiments and evaluations

In this section, we qualify the efficiency and accuracy for NVM Streaker and Fast Calculation. Flexible and varied experiments with Spark big-data benchmarks [46] demonstrate that these methods can work with low complexity, fast simulation speed and high accuracy. It is efficient to analyze the performance variation for big-data researches with these methods under the scenario of NVM.

There are a wide variety of researches on NVM's simulations. Specific methods have significant differences on performance [15, 34]. In this section, evaluations are designed and conducted from two aspects to demonstrate the low simulation costs and high accuracy for NVM Streaker.

First, we use real Spark applications to verify the functionality and the accuracy in performance simulation for NVM Streaker. Based on the design concept, a user application should obtain similar AMAC values or performance values to their most similar imitators during co-running with disturber set. Our experiments with Spark certify that NVM Streaker can lead to similar performance with the average error rate ranging from 2.3–8.8% (Sect. 5.3).

Second, we contrast between real execution results with NVM Streaker and the estimated results with Fast Calculation for these applications. Evaluations illustrate that the simulation methods with these two methods are highly consistent (the average error rate is about 3.6%). Both these two methods can work for performance simulation on NVM efficiently (Sect. 5.4).

Moreover, we contrast the simulation costs between NVM Streaker and a well-known cycle-level simulator, GEM5 + DRAMSim2 [48, 49] to show its low overheads (Sect. 5.5).

5.1 Platform

NVM Streaker is based on a dual-socket, 12-core Intel® Xeon® X5650 server. Prefetching is disabled. Each CPU supports three DDR3 channels. The server has a total of 32 GB memory, which is divided equally between CPUs initially. On this server, there are 10 CPU frequency levels, 2 memory frequency levels and 3 memory channel levels. Based on observations in Sect. 4.1.1, experiments under each parameter configuration follow the common rules. In this section, we choose $\left[\text{DRAM}^{\langle 2.67,1.33,3\rangle}, \text{NVM}^{\langle 2.67,1.33,1\rangle}\right]$ as our platform for evaluations with Spark benchmarks.

5.2 Benchmarks and matrix for accuracy

We evaluate NVM Streaker with three Spark benchmarks [46]. As listed in Table 2, PageRank is a popular algorithm for website. BFS is one of the most popular algorithms for graph or tree traversing. CC is an algorithm to compute the connection. NVM Streaker is evaluated from two aspects, either different size of data input or periodical behavior (kernel time or full execution time). In this subsection, the default runtime configurations for Spark benchmarks are adopted. All the measured AMAC values are normalized to the native DRAM access costs. We use 3 graphs, dolphins.txt, diseasome.txt and p2p-nutella30.txt, as the original data set for our experiments as in Table 2. We develop a group of inputs which may be different either in the data size or in the vertex from these 3 graphs. Consider BFS in Table 2, for example, B_Data^1 and B_Data^2 stand for two graphs which have the same size of 300M while different vertex data. For each benchmark, we experiment with 2–3 different inputs because

Table 2 Benchmarks for qualification of NVM Streaker

Benchmarks	Description	Test type	Original graph	Data input
PageRank	It is a way of measuring the importance of website pages by counting the number and quality of links to a page.	Kernel execution Full execution	Dolphins.txt Graph: ⟨62 node, 318 edge⟩	P_Data^0 P_Data^1
BFS	It is an algorithm for traversing or searching tree or graph data structures.	Full execution	Diseasome.txt Graph: ⟨1419node, 3926edge⟩	B_Data^0 (160 M) B_Data^1 (300 M) B_Data^2 (300 M)
CC	It is an algorithm to compute the connected components.	Full execution	p2p-nutella30.txt Graph: ⟨6682node, 88328edge⟩	C_Data^0 (160 M) C_Data^1 (300 M) C_Data^2 (300 M)

these applications present different behaviors when the data input varies, i.e., different graphs lead to different performance.

As to experiments with NVM Streaker, we measure $\text{AMAC}_{\text{simulated}}$ and $P_{\text{simulated}}$ for applications. These two values are obtained through real execution with NVM Streaker.

As to Fast Calculation, we measure $\text{AMAC}_{\text{estimated}}$ and $P_{\text{estimated}}$ which are obtained through calculation with Eqs. (3) and (4).

The accuracy of NVM Streaker is certified through contrasts between $P_{\text{simulated}}$ values of training sets and real benchmarks. As we mentioned in previous Sect. 2.2, for big-data applications, we use Eq. (5) for error calculation since it is difficult to separate memory instructions from other instructions. As shown in Eq. (5), for a Spark benchmark, $P^{\text{spark}}_{\text{simulated}}$ is the normalized performance of Spark benchmark. $P^{\text{imitator}}_{\text{simulated}}$ is the normalized value of the imitator which has the most similar memory property to the Spark benchmark. The contrasts are made with error as in Eq. (5), which is also a common method in many works such as in [27].

$$\text{error}\,(x) = \frac{\left| P^{\text{imitator}}_{\text{simulated}} - P^{\text{spark}}_{\text{simulated}} \right|}{P^{\text{spark}}_{\text{simulated}}} \tag{5}$$

The accuracy of Fast Calculation is certified through contrasts between NVM Streaker and Fast Calculation. The contrasts are made with error as in Eq. (6):

$$\text{error}\,(x) = \frac{|\text{AMAC}_{\text{estimated}} - \text{AMAC}_{\text{simulated}}|}{\text{AMAC}_{\text{simulated}}} \tag{6}$$

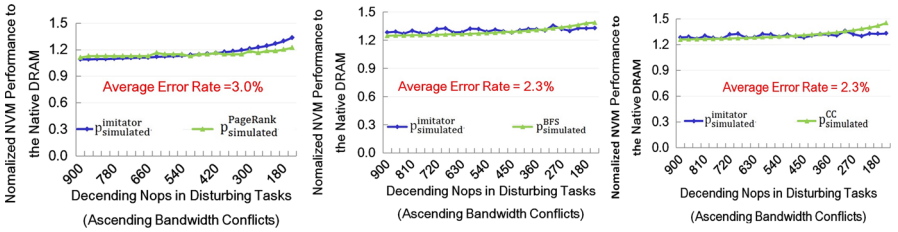

Fig. 9 Evaluation contrast between the Spark benchmarks and the most similar imitators for the accuracy of NVM Streaker. **a** Full execution of PageRank with P_Data^0, **b** BFS with input of B_Data^1, **c** CC with input of C_Data^1

Similar to Eq. (5), we use Eq. (7) for the accuracy of Fast Calculation for big-data applications. It is the error rate between the simulated value with NVM Streaker and the estimated value with Fast Calculation.

$$\text{error}\,(x) = \frac{\left| P^{\text{spark}}_{\text{estimated}} - P^{\text{spark}}_{\text{simulated}} \right|}{P^{\text{spark}}_{\text{simulated}}} \tag{7}$$

5.3 Evaluations for NVM Streaker

In this subsection, we contrast the performance between the Spark benchmark and its most similar imitator task. All experiments are conducted under the hardware environment of $\left[\text{DRAM}^{\langle 2.67,1.33,3\rangle}, \text{NVM}^{\langle 2.67,1.33,1\rangle}\right]$. In these processes, the accuracy of NVM Streaker can be illuminated by the gap (the error rate) between $P_{\text{simulated}}$ of these two kinds of tasks. For example, we can obtain a performance curve for PageRank after co-running it with the disturber set. Then we compare this curve with that of the imitator which has the most similar memory behaviors with PageRank. The consistency of these two curves determines the accuracy of NVM Streaker, i.e., whether user can build simulated NVM environment according to its most similar imitator.

Figure 9 illustrates the contrasts between the Spark benchmarks and their most similar imitators. In these experiments, the Spark benchmarks and their most similar imitators are set to 12-thread with hyperthread. Experiments certificate the feasibility of constructing NVM environment with the most similar imitator of an application. In each figure of Fig. 9, x-axis stands for all disturbers in the co-running process. Y-axis stands for the normalized $P_{\text{simulated}}$ to the DRAM performance. More specifically, the blue curve stands for the normalized $P^{\text{imitator}}_{\text{simulated}}$ during co-running with all disturbers as x-axis shows. The green curve stands for the normalized $P^{\text{spark}}_{\text{simulated}}$ during co-running with all disturbers.

In Fig. 9a, the full execution of PageRank with P_Data^0 has a LLC of about 0.3%, while WR is about 25%. Its most similar imitator task has a LLC of about 0.3%, while WR is about 30%. We can observe that the simulation results of the PageRank are

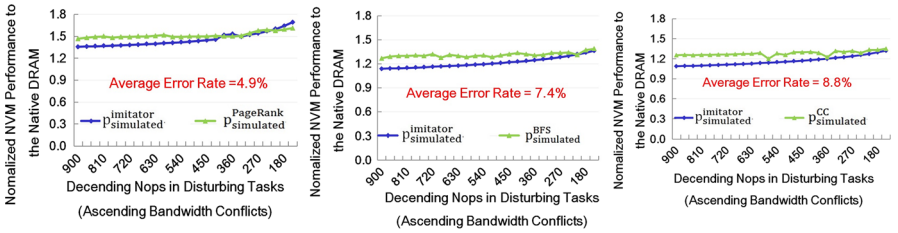

Fig. 10 Extra evaluation contrasts between the Spark benchmarks and the most similar imitators with different data inputs for the accuracy of NVM Streaker. **a** Full execution of PageRank with P_Data[1], **b** BFS with input of B_Data[2], **c** CC with input of C_Data[2]

consistent with those of its most similar imitator. The average error rate is about 3.0%. In Fig. 9b, BFS with input of BFS_Data[1] has a LLC of about 0.3%, while WR is about 17%. Its most similar imitator task has a LLC of about 0.3%, while WR is about 15%. The average error rate is about 2.3%. In Fig. 9c, CC with input of CC_Data[1] has similar performance to BFS. It has a LLC of about 0.3%, while WR is about 17%. Its most similar imitator task has a LLC of about 0.3%, while WR is about 15%. The average error rate is about 2.3%.

For many graph-dealing applications such as the Spark benchmarks in our work, different graph inputs can lead to distinct program behavior. We carry out an extra set of experiments to certify the accuracy of NVM Streaker with another set of data input set. These sets have the same size, e.g., still 300M for BFS and CC, while different data for all these Spark benchmarks as in Fig. 9. Figure 10 demonstrates these extended experiments. In Fig. 10a, the full execution of PageRank with P_Data[1] has a LLC of about 0.3%, while WR is about 33%. Its most similar imitator task has a LLC of about 0.3%, while WR is about 37%. The average error rate is about 4.9%. In Fig. 10b, BFS with input of BFS_Data[2] has a LLC of about 0.3%, while WR is about 21%. Its most similar imitator task has a LLC of about 0.3%, while WR is about 23%. The average error rate is about 7.4%. In Fig. 10c, CC with input of CC_Data[2] has a LLC of about 0.3%, while WR is about 21%. Its most similar imitator task has a LLC of about 0.3%, while WR is about 23%. The average error rate is about 8.8%.

The consistency in the performance curves of the Spark benchmarks and their most similar imitators shows that NVM Streaker is reliable for NVM simulation. It is feasible to build NVM environment according to a user application's most similar imitator. We also observed that, for each different servers, the appropriacy of both the imitator set and the disturber set decide the accuracy of NVM Streak. We will continue to improve and refine the imitator set for more accurate simulation in our future work.

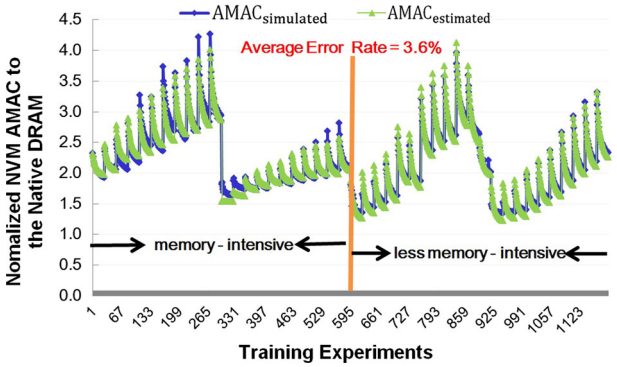

Fig. 11 Contrast between the executed data and the estimated data

5.4 Evaluations for Fast Calculation based on NVM Streaker

In this subsection, we certify the accuracy of Fast Calculation through contrast between NVM Streaker and Fast Calculation.

Fast Calculation devotes to a much faster and easier way for NVM simulation. As introduced in Sect. 4.2, it is an estimation equation for AMAC or performance formed from all the training results on a server. We carry our evaluation experiments under the hardware environment of $\left[\text{DRAM}^{\langle 2.67,1.33,3\rangle}, \text{NVM}^{\langle 2.67,1.33,1\rangle}\right]$. Equation (8) is extracted from the training process introduced in Sect. 4.1.2:

$$\begin{aligned}\text{AMAC}_{\text{estimated}} = 3.829 &+ \left(-0.164 \cdot \frac{\text{RLLC}_0}{\text{TH}_0} - 0.166 \cdot \frac{\text{WLLC}_0}{\text{TH}_0}\right) \\ &+ \left(-0.047 \cdot \frac{\text{RLLC}_0}{\text{TH}_0} - 0.052 \cdot \frac{\text{WLLC}_0}{\text{TH}_0} + 0.72\right) \cdot \text{RLLC}_1 \end{aligned} \tag{8}$$

With thorough training process, Fast Calculation can work as accurately as NVM Streaker. It is more efficient because it does not require real execution process. Figure 11 contrasts $\text{AMAC}_{\text{simulated}}$ and $\text{AMAC}_{\text{estimated}}$ between NVM Streaker and Fast Calculation with all disturbers and imitators which are introduced in Sect. 4.1.1. We collect thousands of experiment data in the figure as x-axis shows. The blue curve stands for all $\text{AMAC}_{\text{simulated}}$ with NVM Streaker. The green curve stands for all $\text{AMAC}_{\text{estimated}}$ with Eq. (8) of Fast Calculation. In the figure, data are separated into two regions according to the degree of LLC miss rate as discussed in Sect. 4.1.1.2. We can observe that the average error rate between NVM Streaker and Fast Calculation is about 3.6% (ranging from 0 to 20%). Both these method can work well. With thorough consideration on the training process, the estimation method of Fast Calculation can meet the research requirements efficiently without real execution process.

Figure 11 also demonstrates the simulation scope in our work. All the experiments show that AMAC ranges from 1.4x to 4.3x for both NVM Streaker and Fast Calculation.

We apply the contrast experiment on big-data Spark applications with $\text{TH}_0 = 12$. Figure 12 contrasts $P_{\text{simulated}}$ and $P_{\text{estimated}}$ between NVM Streaker and Fast Calcu-

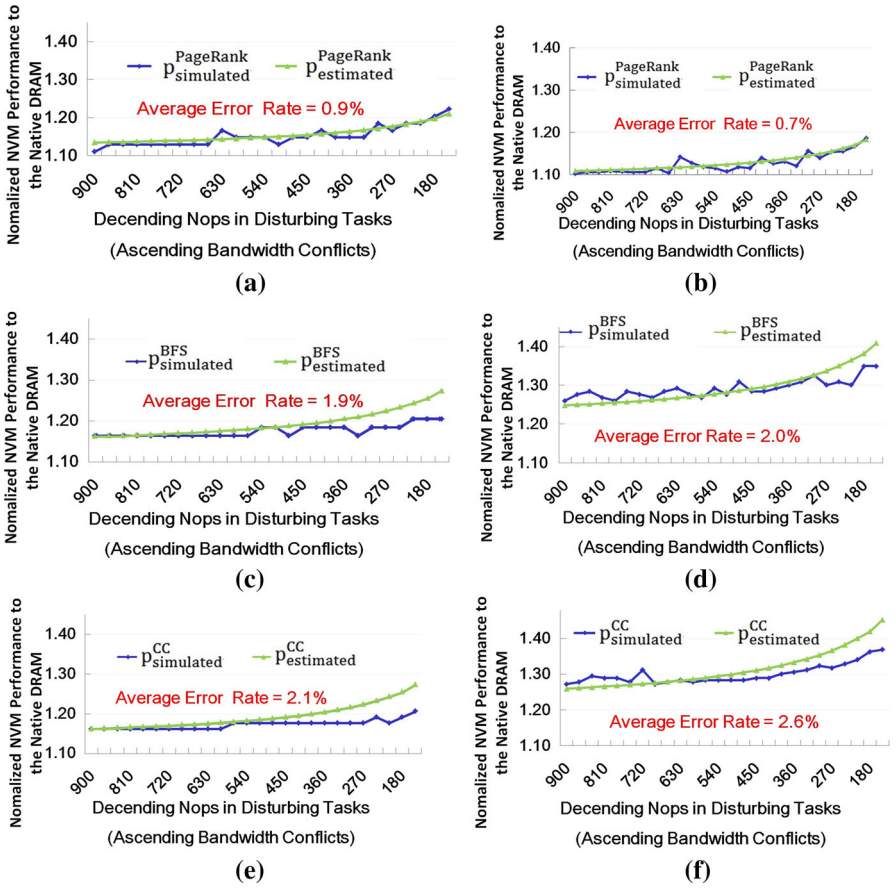

Fig. 12 Contrasts between NVM Streaker and Fast Calculation with Spark benchmarks. **a** Full execution of PageRank with P_Data^0, **b** Kernel stage of PageRank with P_Data^0, **c** BFS with input of BFS_Data^0, **d** BFS with input of BFS_Data^1, **e** CC with input of CC_Data^0, **f** CC with input of CC_Data^1

lation for PageRank, BFS and CC. In each figure, the blue curve stands for $P_{\text{simulated}}$ values and the green curve stands for the $P_{\text{estimated}}$ values. Each benchmark is investigated from two aspects as listed in Table 2.

Figure 12a, b demonstrates the contrasts between $P_{\text{simulated}}$ and $P_{\text{estimated}}$ for PageRank. Figure 12a shows the contrast for the full execution time with input of P_Data^0. Among all memory operations, the WR is about 33%. The $P^{\text{PageRank}}_{\text{simulated}}$ ranges from 1.102 to 1.167. The $P^{\text{PageRank}}_{\text{estimated}}$ ranges from 1.109 to 1.170. The error rate between the two curves is about 0.9%. Figure 12b shows the contrast for the PageRank kernel stage time with input of P_Data^0. Among all memory operations, the WR is about 17%. The $P^{\text{PageRank}}_{\text{simulated}}$ ranges from 1.111 to 1.204. The $P^{\text{PageRank}}_{\text{estimated}}$ ranges from 1.135 to 1.198. The average error rate is about 0.7%.

Figure 12c, d demonstrates the contrasts with different data set for BFS. Figure 12c shows the contrast for BFS with **input of BFS**_Data^0. Among all memory operations,

Table 3 Contrasts on the simulation costs between NVM Streaker and DRAMSim2

Input / Simulation cost(s) / Benchmark	dolphins.txt Graph:<62 node, 318 edge>		diseasome.txt Graph:<1419node,3926edge>		p2p-nutella30.txt Graph:<6682node,88328edge>	
	NVM Streaker	GEM5+DRAMSim2	NVM Streaker	GEM5+DRAMSim2	NVM Streaker	GEM5+DRAMSim2
PageRank	0.003	9.9	0.004	106.6	0.047	2421
BFS	0.002	1.9	0.002	1.9	0.002	1.88
CC	0.002	7.3	0.003	75	0.042	2148.7

the WR is about 17%. The $P^{\text{BFS}}_{\text{simulated}}$ ranges from 1.163 to 1.204. The $P^{\text{BFS}}_{\text{estimated}}$ ranges from 1.162 to 1.273. The error rate between the two curves is about 1.9%. Figure 12d shows the contrast for BFS with **input of BFS**_Data[1]. Among all memory operations, the WR is about 21%. The $P^{\text{BFS}}_{\text{simulated}}$ ranges from 1.260 to 1.350. The $P^{\text{BFS}}_{\text{estimated}}$ ranges from 1.248 to 1.409. The average error rate is about 2.0%.

Figure 12e, f demonstrates the contrasts with different data set for CC which leads to different memory properties. Figure 12e shows the contrast for BFS with input of CC_Data[0]. Among all memory operations, the WR is about 14%. The $P^{\text{CC}}_{\text{simulated}}$ ranges from 1.162 to 1.206. The $P^{\text{CC}}_{\text{estimated}}$ ranges from 1.162 to 1.273. The average error rate is about 2.1%. Figure 12f shows the contrast for CC with input of CC_Data[1]. Among all memory operations, the WR is about 21%. The $P^{\text{CC}}_{\text{simulated}}$ ranges from 1.272 to 1.370. The $P^{\text{CC}}_{\text{estimated}}$ ranges from 1.26 to 1.453. The error rate between the two curves is about 2.6%.

All the above experiments show that NVM Streaker and Fast Calculation can produce highly consistent simulation results. With these two methods, users can configure the NVM environments flexibly according to their needs. We also observed that these benchmarks are relatively less memory-intensive and they have lower write miss ratios. We are going for a further study for extending NVM Streaker and Fast Calculation to more hardware platforms and more big-data applications.

5.5 Extended evaluation on time costs

There are many NVM-based simulators [4, 8, 26, 27]. However, different design details have significant differences on performance [45]. It is difficult to compare NVM Streaker with physical design targeted simulators. We illustrate the significant difference in the simulation costs with a well-known cycle-level simulator, GEM5 + DRAMSim2. Table 3 shows the contrasts between these two simulation techniques. The details of the benchmark and the input sets are included in the table. Experiments show that NVM Streaker is much time-saving than the cycle-level simulator, with almost negligible simulation costs.

6 Related works

NVM in the homogeneous architecture or the hybrid architecture has been explored in many contexts, e.g., for database [6], for storage system [13, 19, 41, 42], and for

main-memory system [14, 28, 29]. Many work has investigated new materials, SSD [13, 25, 31] or PCM [14, 15, 24, 28, 29, 45], in performance, capacity and endurance. However, modeling NVM latency is still challenging because of the diversity and commercial unavailability of these architectures.

6.1 Simulator techniques

There are various techniques on NVM, such as NVSim [8], NVMain [26, 27], LightNVM [4]. These efforts were helpful in the hardware design for different NVMs. With details of memory cells, these work could calculate or evaluate the memory latency and bandwidth for the target architecture. Dong et al. [6] proposed a circuit-level model, NVSim, for NVM performance, energy, and area estimations based on CACTI [38]. Bjorling et al. [4] introduced LightNVM which comprises both a simulated memory-backed storage for IO and a hardware simulation for accurate NVM timings. Poremba et al. [26, 27] made a continuous development on the memory simulator of NVMain which include a thorough consideration for the simulation speed, the sub-array-level parallelism, the distributed energy profiling. Dulloor et al. proposed a configurable simulator, PM Emulator [9] and necessary OS management for hybrid memory architecture.

Some researches relied on specified hardware support. HMEP [20] built their framework with special CPU microcode. HASTE [5] needed four FPGAs, eight memory controller and significant OS modifications simultaneously. Malicevic et al. [20] proposed HEMP for NVM simulation. This simulator simulates only the average latencies and not NVM's device-specific characteristics. These hardware simulators always target at specified memory architectures, and lack commercial availability for further researches. Sengupta et al. [30] implemented a software-based emulation solution that injects a pre-computed delay in the stream of memory reference to achieve NVM latency.

6.2 Optimization techniques

Optimizations on NVMs are hot topics for lower power consumption, higher performance, and longer lifetime [12, 14, 16, 17, 33–36]. Among all NVM materials, PCM is more competitive in performance and attracts more interests [14, 28, 29, 45]. Qureshi et al. [28] proposed a hybrid main-memory organization corresponding optimizations for both performance and longer endurance. It was among the first architectural study that proposes and evaluates PCM for main-memory systems. In their later work [29], the researches improved their work on the problem of longer write latency. Lee et al. [14] proposed architectural enhancements for using PCM as a DRAM alternative. This work dipped into the optimizations on write operations via PCM buffer organizations and partial write. Dhiman et al. [7] proposed a hybrid PCM and DRAM main-memory architecture. This work focused on reduction of the energy consumption. Zhou et al. [45] devoted to enhance the endurance of PCM with two optimizations on writes, redundant bit-writes and wear leveling, which can dramatically extend the lifetime of PCM.

Tseng et al. [32] and Hu et al. [10] made a series effort on reduction on the memory accesses costs. They proposed an ILP formulation optimization and a heuristic-guided concatenation scheduling method for NVM architectures which can reduce the high NVM access costs. In their later work [10], they further reduced unnecessary writing operations through reasonable scheduling, and data recomputation, etc. All these methods need to analyze the specific situation of the program at compile time, and software controllable cache.

Malicevic et al. [16] focused on the replaceability of NVM to DRAM. They investigated NVM in meeting the memory demands of graph analytics frameworks on HEMP. Agarwal et al. [3] studied the BW maximizing page placement policies for hybrid memory system which are composed of CPU memory and GPU memory.

The development of NVM has stimulated the researches in system software supports. The work in [22] had foreseen the key issues which the hybrid memory system brings. It has primarily explored the necessary supports from the operating system for NVM, which included basic hardware and software designs, page type, file type.

6.3 Memory management techniques

Persistent memory (PM) system, storage-class memory (SCM) and persistent languages focus more new management techniques in order to realize data object persistence [34], or language level persistence [1–3, 5]. User applications can utilize these techniques through customized interfaces for NVM, which impose higher requirements on developers. Wang et al. [35] were one of the forward-looking efforts which have devoted to compiler optimizations for hybrid memory architectures. They proposed more reasonable memory management strategies for better data layout. NVM Streaker worked as one of the experimental platforms in their researches.

For any research with NVM, it is important to simulate and analyze the optimizations in a feasible way. Both the complexity and the exactness are necessary for the corresponding researches. NVM Streaker can meet these demands and make it possible for the combination research with big-data applications.

7 Conclusions and future work

In this work, we introduce a fast and reconfigurable simulation method, called NVM Streaker, for NVM-based memory architectures. NVM Streaker can realize a fast simulation via the direct hardware parameter configuration and a software disturbing mechanism. Moreover, it has high reconfigurability which enables users to configure various NVM environments according to their needs. Experiments with Spark applications show that NVM Streaker's low complexity and high configurability.

The characteristic of low cost makes NVM Streaker competent for performance analyses in many prospective studies which orient to future hybrid memory architectures. NVM Streaker still needs a further effort to minimize the error rate for

write-intensive applications. Our next steps include enhancing NVM Streaker for write-intensive applications, and extending it to more real big-data applications and more hardware platforms.

References

1. Atkinson MP, Bailey PJ, Chisholm KJ et al (1983) An approach to persistent programming. Computer Journal 26(4):360–365
2. Atkinson MP, Daynes L, Jordan MJ et al (1996) An orthogonally persistent java. SIGMOD Rec 25(4):68–75
3. Agarwal N, Nellans D, Stephenson M et al (2015) Page placement strategies for GPUs within heterogeneous memory systems. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, pp 607–618
4. BjÃÿrling M, Madsen J, Bonnet P et al (2014) LightNVM: lightning fast evaluation platform for non-volatilememories. In: 5th Annual Non-Volatile Memories Workshop
5. Caulfield AM, Coburn J, Mollov T et al (2010) Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 13–19 Nov 2010, pp 1–11
6. DeBrabant J, Arulraj J, Pavlo A et al (2014) A prolegomenon on OLTP database systems for non-volatile memory. In: ADMS@VLDB, pp 57–63
7. Dhiman G, Ayoub R, Rosing T (2009) PDRAM: a hybrid PRAM and DRAM main memory system. In: Proceedings of the 46th Annual Design Automation Conference (DAC'09), pp 664–669
8. Dong X, Jouppi N, Xie Y (2012) Nvsim: a circuit-level performance, energy, and area model for emerging nonvolatile memory. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp 994–1007
9. Dulloor SR, Kumar S, Keshavamurthy A et al (2014) System software for persistent memory. In: Proceedings of EuroSys
10. Hu J, Xue C, Tseng W et al (2010) Minimizing write activities to non-volatile memory via scheduling and recomputation. In: Proceedings of 2010 IEEE 8th Symposium on Application Specific Processors (SASP), pp 101–106
11. Hu J, Xue C, Zhuge Q, Tseng WC et al (2011) Towards energy efficient hybrid on-chip scratch pad memory with nonvolatile memory. In: Proceedings of Design, Automation Test in Europe Conference and Exhibition (DATE), Mar 2011, pp 1–6
12. Hu J, Xue CJ et al (2012) Scheduling to optimize cache utilization for non-volatile main memories. In: IEEE Transactions on Computers (TC), Dec 2012
13. Hu X, Eleftherou E, Haas et al R (2009) Write amplification analysis in flash-based solid state drives. In: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference
14. Lee B, Ipek E, Mutlu O et al (2009) Architecting phase change memory as a scalable dram alternative. In: Proceedings of the 36th International Symposium on Computer Architecture, 2009
15. Lee BC, Ipek E, Mutlu O, Burger D (2007) Architecting phase change memory as a scalable DRAM alternative. In Proceedings of ISCA 36, June 2007
16. Liu D, Zhong K, Wang T, Wang Y, Shao Z, Sha EHM, Xue J (2016) Durable address translation in PCM-based flash storage systems. In: IEEE Transactions on Parallel and Distributed Systems (TPDS)
17. Li Q, Zhao Y, Hu J et al (2012) MGC: multiple graph-coloring for non-volatile memory based hybrid scratchpad memory. In: Proceedings of 16th Workshop on Interaction Between Compilers and Computer Architectures (INTERACT), Feb 2012, pp 17–24
18. Liu T, Zhao Y, Xue C et al (2011) Power-aware variable partitioning for DSPS with hybrid pram and dram main memory. In: Proceedings of 48th ACM/EDAC/IEEE Design Automation Conference (DAC), June 2011, pp 405–410
19. Lu Y, Shu J, Zheng W (2013) Extending the lifetime of flash-based storage through reducing write. In: The 11th USENIX Conference on File and Storage Technologies

20. Malicevic J, Dulloor S, Sundaram N et al (2015) Exploiting NVM in large-scale graph analytics. In: INFLOW'15
21. Mars J, Tang L, Hundt R et al Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In: Proceedings of Micro'11, Dec 2011, pp 248–259
22. Mogul JC, Argollo E, Shah M et al (2009) Operating system support for NVM + DRAM hybrid main memory. In: Hot Topics in Operating Systems
23. Manchanda N, Anand K (2010-05-04) Non-Uniform Memory Access (NUMA). http://cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf. New York University. Retrieved 27 Jan 2014
24. Mangalagiri P, Sarpatwari K, Yanamandra A et al (2008) A low-power phase change memory based hybrid cache architecture. In: Proceedings of 18th ACM Great Lakes Symposium on VLSI, pp 395–398
25. Oh Y, Choi J, Lee D et al (2013) Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In: Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12), pp 25–25
26. Poremba M, Xie Y (2012) Nvmain: an architectural-level main memory simulator for emerging non-volatile memories. In: Proceedings of IEEE Computer Society Annual Symposium on VLSI, Aug 2012, pp 392–397
27. Poremba M, Zhang T, Xie Y (2015) NVMain 2.0: a user-friendly memory simulator to model (non-)volatile memory systems. Comput Archit Lett 14(2):140–143
28. Qureshi M, Srinivasan V, Rivers JA (2009) Scalable high performance main memory system using phase-change memory technology. In: Proceedings of the 36th International Symposium on Computer Architecture, 2009
29. Qureshi MK, Franceschini MM, Lastras-montao LA (2010) Improving read performance of phase change memories via write cancellation and write pausing. In: Proceedings of 16th International Symposium on High-Performance Computer Architecture, 2010
30. Sengupta D, Wang Q et al (2015) A framework for emulating non-volatile memory systems with different performance characteristics. In: Proceedings of ICPE'14, 2015
31. Soundararajan G, Prabhakaran V, Balakrishnan M et al (2010) Extending SSD lifetimes with disk-based write caches. In: Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10), pp 8–8
32. Tseng W, Xue C, Zhuge Q et al (2010) Optimal scheduling to minimize non-volatile memory access time with hardware cache. In: Proceedings of 18th IEEE/IFIP VLSI System on Chip Conference (VLSI-SoC), pp 131–136
33. Uttamchandani S (2015) Scale out storage architectures in the NVM Era, evolution or revolution? Flashmemory Summit, Santa Clara, CA
34. Volos H, Tack AJ, Swift MM (2011) Mnemosyne: lightweight persistent memory. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, March 05–11, 2011, Newport Beach, California, USA https://doi.org/10.1145/1950365.1950379
35. Wang C, Cao T, Zigman J, Lv F, Zhang Y, Feng X (2016). Efficient management for hybrid memory in managed language runtime. In: Proceedings of Network and Parallel Computing
36. Wang D, Ganesh B, Tuaycharoen N et al (2005) DRAMsim: a memory system simulator. SIGARCH Comput Archit News 33(4):100–107
37. Wei W, Jiang D, Chen M (2014) Exploring opportunities for non-volatile memories in big data applications. In: Big Data Benchmarks, Performance Optimization, and Emerging Hardware. Springer, Berlin, pp 209–220
38. Wilton SJE, Jouppi NP (1996) CACTI: an enhanced cache access and cycle time model. IEEE J Solid-State Circuits 31(5):677–688
39. Wu X, Li J, Zhang L, Speight E et al (2009) Power and performance of read-write aware hybrid caches with non-volatile memories. In: Design, Automation and Test in Europe Conference and Exhibition, IEEE, pp 737–742
40. Wu X, Li J, Zhang L, Speight E et al (2009) Hybrid cache architecture with disparate memory technologies. SIGARCH Comput Archit News 37(3):34–45. https://doi.org/10.1145/1555815.1555761=0pt
41. Wang Y, Wang T, Liu D, Shao Z, Xue Jingling (2017) Fine grained, direct access file system support for storage class memory. J Syst Archit 72:80–92

42. Wang Y, Wang T, Shao Z, Liu D, Xue J (2015) File system-independent block device support for storage class memory. In: The International Workshop of Software-Defined Data Communications And Storage (SDDCS) 2015, in Conjunction with IEEE INFOCOM 2015, Hongkong
43. Xue C, Zhang Y, Chen Y et al (2011) Emerging non-volatile memories: opportunities and challenges. In: Proceedings of 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES +ISSS), Oct 2011, pp 325–334
44. Yang H, Breslow A, Mars J et al (2013) Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In: International Symposium on Computer Architecture 2013
45. Zhou P, Zhao B, Yang J et al (2009) A durable and energy efficient main memory using phase change memory technology. In: Proceedings of the 36th International Symposium on Computer Architecture
46. http://spark.apache.org/
47. https://www.mathworks.com/help/matlab/getting-started-with-matlab.html
48. http://www.m5sim.org/Main_Page
49. https://eng.umd.edu/~blj/dramsim/