International Journal of Parallel Programming
https://doi.org/10.1007/s10766-024-00776-x

Intelligent Page Migration on Heterogeneous Memory by Using Transformer

Songwen Pei[1,2] · **Wei Qin**[1] · **Jianan Li**[1] · **Junhao Tan**[1] · **Jie Tang**[3] · **Jean-Luc Gaudiot**[4]

Received: 13 February 2023 / Accepted: 17 August 2024

Abstract

Locality-based migration strategies are widely used in existing memory space management. Such type of strategies are consistently confronts with challenges in efficiently managing pages migration within constrained memory space, especially when new architecture such as hybrid of DRAM and NVM are emerging. Here we propose TransMigrator, an innovative predictive page migration model based on transformer architecture, which obtains a qualitative leap in the breadth and accuracy of prediction compared with traditional local-based methods. TransMigrator utilizes an end-to-end neural network to learn memory access behavior and page migration record in the long-term history and predict the most likely next page to fetch. Furthermore, a migration-management mechanism is designed to support the page-feeding from predictor, which in another way enhance the model robustness. The model achieves an average prediction accuracy better than 0.72, and saves an average of 0.24 access time overhead compared to strategies such as AC-CLOCK, THMigrator, and VC-HMM.

✉ Songwen Pei
swpei@usst.edu.cn

✉ Wei Qin
201440056@st.usst.edu.cn

1 University of Shanghai for Science and Technology, Shanghai 200093, China

2 State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China

3 South China University of Technology, Guangzhou 510641, China

4 University of California, Irvine, CA 92617, USA

Published online: 12 September 2024

1 Introduction

Driven by the growth of the data-intensive applications such as big data computing, cloud computing and super high-intensity training work, the traditional computing mode has been making the shift to large-scale data processing. The demand of ability to process large-scale data is not only dependent on computational power, but also on the capability of memory to store data. The Limitations such as unscalable capacity and huge static energy consumption of traditional DRAM system, make it hard to respond to the new challenges. A research [9] has revealed that DRAM consumes a portion 30–48% of the total system's energy. Nowadays, data centers are increasingly focused on memory capabilities, efficiency and capacity of memory systems are becoming as critical as computational power. In response to the limitations of DRAM, Non-Volatile Random-Access Memory (NVM) has emerged as an alternative. which offers higher storage density and lower energy consumption [5, 6, 24]. Despite its benefits, NVM is also facing challenges. It generally has lower performance compared to DRAM, like slower data access speed, particularly in writing, limited write endurance, and higher write energy consumption. Being specific, Phase-Changing Memory (PCM) requires more energy to change between amorphous and crystalline state compared to the energy needed for moving electric charges in the transistor [2]. Additionally, the PCM's write latency can be up to 10 times higher than that of DRAM.

Due to its inherent limitations, current NVM cannot completely replace DRAM entirely. As an alternative to a complete replacement, the concept of heterogeneous memory system has been proposed which integrates both DRAM and NVM. aiming to leverage the advantages of each while mitigating their disadvantages. [7, 18, 28]. To improve system performance, "Hot pages," which are accessed more frequently, should ideally be placed in DRAM due to its faster access times. Conversely, "cold pages," which are accessed less frequently, are better suited for NVM, which offers higher storage density and lower energy consumption.

There are generally two organizational approaches for hybrid memory systems: vertical and horizontal as shown in Fig. 1. In the vertical organization, DRAM is used as a cache for NVM. DRAM acts as a smaller, faster storage layer that holds frequently accessed data, while NVM serves as a larger, slower storage layer. When a request is missed in DRAM, the system will retrieve the data from NVM. The use of NVM is transparent to the programmer in Memory Mode. The vertical organization also allows for the option to bypass DRAM and access NVM directly, such as the strategy THMigrator [19].

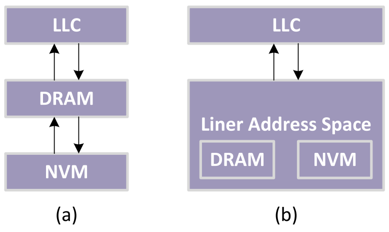


Fig. 1 Two kinds of hybrid memory organizations. Vertical organization in (**a**) and horizontal organization in (**b**)

In the horizontal organization, DRAM and NVM are combined to form a single address space. In some strategies, a page could only reside in one memory device, such as CLOCK-DWF [14] and AC-CLOCK [13], of which type is named exclusive strategy. In contrast, a page is allowed exist in both memory devices simultaneously, such as APMigrate [27] and VC-HMM [22], which are referred to as a redundant strategy. The vertical organization naturally falls under the redundant strategy category, because a page could be hold both in DRAM and NVM.

In order to take advantage of the hybrid memory system, pages should be placed in DRAM and NVM carefully through hotness evaluation. CLOCKDWF migrates only dirty pages (*i.e.*, pages being written) to DRAM, while AC-CLOCK migrates pages being both read and written, which aims to keep frequently accessed pages in the faster DRAM. THMigrator uses a fixed threshold to decide when to migrate pages, which is not flexible enough to adapt to the varying access patterns of different applications. APMigrate and VC-HMM try to ease the inflexibility of fixed thresholds by dynamically adjusting the threshold based on the benefit derived from previous migrations. But the initial value of the threshold makes the system fragile. The initial value of the threshold is crucial. If set too low, the system may perform many migrations that make little contribution to performance. These inefficient migrations may go unnoticed until the pages are eventually evicted. However, when pages are migrated back to NVM, the threshold may become overly large, preventing further beneficial migrations from being executed. This pitfall is demonstrated in our experiments.

The page prediction is crucial in migrating strategy, and we found that neural networks are particularly well-suited for this task because they can handle complex, nonlinear relationships in data. They can also adapt to changes in access patterns over time, making them a dynamic solution for memory management.

There are some Existing Neural Network Approaches in page migration in hybrid memory system. Kleio [10] trains a separate neural network for each important pages. Each network predicts the access count for its respective page in the next time interval. This method, while potentially accurate, can be complicated and resource-intensive due to the individual networks for each page. DeepSwapper [3] uses Long Short-Term Memory (LSTM) networks, a type of recurrent neural network, to predict the next memory access sequence based on previous sequence. It operates on the differences between adjacent addresses rather than the addresses themselves, which can lead to instability and low precision when recovering the actual addresses. In contrast to the aforementioned methods, TransMigrator [20] is an extension of previous work that a single, end-to-end neural network to predict hot pages directly. This approach aims to simplify the system design by using one unified network instead of multiple individual networks. In addition to utilizing the page access history as transformer input, this paper also integrates the records of page migration and eviction to enhance the training data, thereby improving the predictive accuracy. Furthermore, the paper redesigns the page migration mechanism, optimizing the previous algorithm that determined the migration queue based solely on access history. A more suitable and rational page migration strategy has been constructed, which aligns better with the predictions made by the transformer. Here lists the main contributions as follows:

- A transformer-based end-to-end predictor is proposed for hot page prediction.
- Migration and eviction history are included as network input parameters.
- TransMigrator is proposed as a robust hybrid memory management architecture. This architecture integrates a neural network predictor with a threshold-based fallback strategy. The dual approach ensures access latency losses are kept within acceptable range when predictor fails.
- The migration part of Migrator is reconstructed to assign a higher level of access priority to prediction result, which aligns better which predictor part.

2 TransMigrator

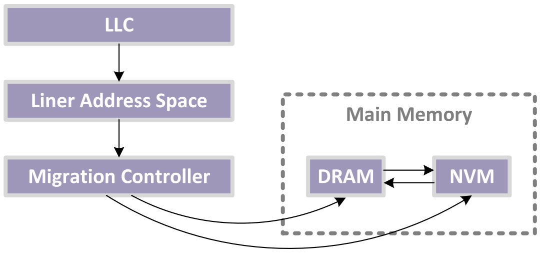
Figure 2 provides a visual representation of how TransMigrator integrates into the overall system architecture. TransMigrator is designed for use in a hybrid memory system with a horizontal organization. The system employs a redundant strategy, meaning that pages can exist in both DRAM and NVM simultaneously. TransMigrator is designed to intercept all Last-Level Cache (LLC) misses, using migration controller (MigC), which allows TransMigrator to make informed decisions about page migration based on actual access patterns. The MigC handles the translation of addresses between the unified address space and the physical locations in either DRAM or NVM, and is responsible for the actual migration of pages between DRAM and NVM.

Figure 3 offers a detailed look at the Migration Controller (MigC) within the TransMigrator system. The left part is responsible for predicting which pages are likely to become hot in the future. It uses a buffer that stores page numbers to prepare for the network input. The right part manages the actual movement of pages between DRAM and NVM based on the predictions.The predictor is a standard transformer model, which is a type of neural network architecture known for its ability to handle sequential data effectively.

There are 3 components composing the right part:

- NVM Page List (NPL) tracks which pages are currently stored in NVM.
- DRAM Page List (DPL) tracks which pages are currently stored in DRAM.
- Candidate Page List (CPL), a list of pages that have the potential to become hot.

Fig. 2 System overview



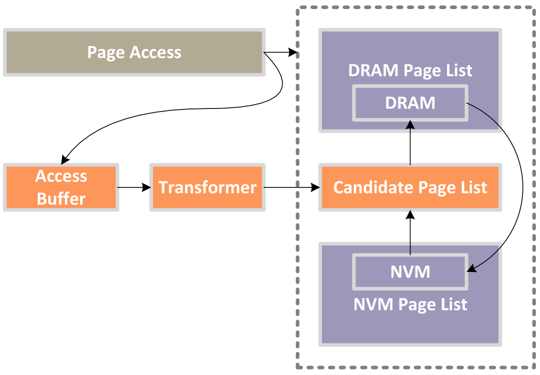


Fig. 3 Details of migration controller

These pages are considered for migration to DRAM upon their next access. Pages in the CPL are subject to a lifetime expiration. If not accessed within their lifetime, they are removed from the CPL. A request counter is used to track the expiration of pages in the CPL. The expiration time for a page is set based on the current counter value plus the predetermined lifetime.

The CPL serves as a bridge between the prediction part and the migration part. It allows the system to prepare migrate pages that the predictor identifies as potentially hot ones.

This design considers the balance between predictive intelligence and traditional threshold-based methods. To address potential inaccuracies in the predictor's performance, a fixed threshold method runs in parallel with the predictor. This dual approach ensures that the system remains robust and can still make effective migration decisions even if the neural network's predictions are not always accurate.

2.1 Design of Neural Network

The network takes a sequence of page numbers as input data, aiming to predict the page number that will be most frequently accessed in the subsequent sequence. The page prediction is formulated as a classification task, where each page number is treated like a 'word' in natural language processing, and each word is mapped to an embedding vector, as shown in the Fig. 4.

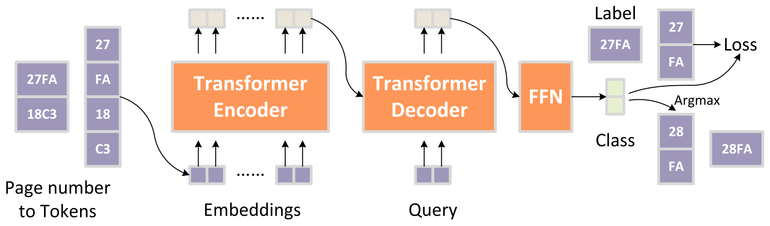


Fig. 4 Details of the neural network

Due to the large address space, it's impractical to map each page number directly to an embedding vector in a dictionary. To address this, page numbers are split into smaller tokens, similar to techniques used in natural language processing. These tokens are of the same length in bits. Such approach of tokenization helps to reduce the dictionary size, making it more manageable to create a mapping from tokens to embedding vectors.

We start the development process by modeling it as a sequence-to-sequence prediction, using the differences in adjacent addresses as input and output. The input was normalized, and the loss function was based on mean square error.

As shown in previous works, recovering the original addresses from the predicted differences proved to be difficult. It was challenging to accurately determine the mean, standard deviation, and base of the differences during the recovery process. Due to the attributes of memory access traces were found to be intrinsically unstable, which makes it difficult for simple regression approaches to work effectively.

Both Long Short-Term Memory (LSTM) networks and transformer models were experimented with as the backbone of the network. At last the transformer model was found to offer better speed and accuracy in handling the task. Based on the evaluation, the transformer model was chosen for its superior performance in terms of both speed and accuracy, making it the ideal candidate for the end-to-end network in TransMigrator.

Input Sequence The network takes as input a sequence of L_{seq} page numbers, which are the addresses of memory pages.

Output Sequence The output of the network is a sequence of N small tokens. These tokens can be reassembled to form the predicted page number.

Tokenization Each page number, which is 20 bits long due to the 32-bit address space and the 12 bits required for the 4 KB page size, is split into N small tokens, each of L_{token} bits. If 20 is not a multiple of L_{token}, zeros are padded to the high bits of the tokens to complete the sequence.

Network In terms of page numbers, the network operates on a many-to-one basis, meaning multiple input page numbers are used to predict a single output page number. However, in terms of tokens, the network is many-to-many, as multiple input tokens are used to predict multiple output tokens. The dimension of the token embeddings and the model in the transformer are the same, denoted as d_{model}. The dimension of the feedforward networks within the transformer is twice the model dimension, *i.e.*, $2 \times d_{model}$.

Transformer Encoder and Decoder The tokens are converted to embeddings, which serve as the input to the transformer encoder. The input to the transformer decoder consists of N learnable embeddings. The features of the decoder output are transformed from d_{model} to C by a linear layer, where C is the dictionary size, calculated as $2^{L_{token}}$. The cross-entropy between the decoder output and the actual labels (the correct tokens) is used as the loss function. This loss is minimized during the training process to improve the accuracy of the network.

2.2 Page Migration

We present a dual-path mechanism in TransMigrator for handling memory access events, involving both the predictor part and the migration part.

Predictor Part: When a memory access occurs, the accessed page number is added to an access buffer. If the buffer reaches its capacity (L pages), all stored page numbers are used as input for the predictor. The output page number from the predictor, which is a prediction of the next hot page, is added to the Candidate Page List (CPL) with a lifetime (L_{seq}).

Migration Part: Pages in DRAM are maintained in LRU (Least Recently Used) order in the DRAM Page List (DPL). Upon access, a page is moved to the head of DPL. If a write operation occurs, the dirty bit is set. If DRAM is full, the tail page of DPL is evicted to NVM. Pages in CPL, if accessed, are copied into DRAM. This approach avoids clean pages write-back, saving NVM write operations. Pages in CPL with expired lifetimes are removed after each access cycle to filter out false positives and reduce unnecessary migrations. Pages in NVM are assigned with counters. When a counter exceeds a threshold, the page is added to CPL with a lifetime (L_{seq}). If a page is not found in DRAM or NVM, a page fault occurs. The page is loaded from external storage to DRAM once sufficient DRAM space has been confirmed.

Migration Procedure: When page P is accessed, its counter is incremented, and its number is added to the access buffer. The MigC searches for page P in DPL, CPL, and NPL in that order.

Operation 1 If P is in DPL, move P to the head and set the dirty bit if it's a write operation.

Operation 2 If P is in CPL, migrate P to DRAM. If DRAM is full, evict the tail page of CPL to NVM, handling the dirty bit and data preservation. Add P to DPL and remove from CPL.

Operation 3 If P is in NPL, increment its counter. If it reaches the threshold, reset the counter, add P to CPL with a lifetime.

Operation 4 If P is not in any list, load it from storage into DRAM after ensuring space, and place it at the head of DPL.

Post-Operation Cleanup 1 MigC removes all pages from CPL that have expired their lifetimes.

Post-Operation Cleanup 2 MigC removes last 3 pages in DPL queue, and transfer them to CPL.

Post-Operation Cleanup 3 Invoke the Predictor function and feed it with most recent operation history information. MigC fetches 3 most likely pages from prediction results, and move them to tail of DPL queue.

Algorithm 1 Page Migration algorithm

```
Require: DPL, NPL, CPL, request counter $c_{req}$, and the threshold
  function OnPageAccess(page p, operation op)
  $c_{req} \leftarrow c_{req} + 1$;
  if $p$ in $DPL$ then                                    ▷ in DRAM
      $node \leftarrow index\ DPL\ by\ p$;
      if $op$ is write then
          $node.dirty \leftarrow true$;
      end if
      move $node$ to $DPL$ head;
  else if $p$ in $CPL$ then                               ▷ in candidate list
      $node \leftarrow index\ CPL\ by\ p$;
      if no free page in DRAM then
          $victim \leftarrow DPL\ tail$;
          MoveToNVM($victim$);
      end if
      remove $node$ from $CPL$;
      add $node$ to $DPL$ head;
  else if $p$ in $NPL$ then                               ▷ in NVM
      $node \leftarrow index\ NPL\ by\ p$;
      $node.count \leftarrow node.count + 1$;
      if $node.count$ >threshold then
          add $node$ to $CPL$ head;
      end if
  else                                                    ▷ page fault
      if no free page in DRAM then
          $victim \leftarrow DPL\ tail$;
          MoveToNVM($victim$);
      end if
      add new page to $DPL$ head;
  end if
  for all $node$ of $CPL$ do                              ▷ check lifetime
      if $node.expireTime >c_{req}$ then
          remove $node$ from $CPL$;
      end if
  end for
  for 3 $node$ of $DPL$ tail do
      remove $node$ from $DPL$;
      add $node$ to $CPL$;
  end for
  Invoke $Predictor()$;
  $node$ 1 $\leftarrow$ $Predictor.result$ 1;
  $node$ 2 $\leftarrow$ $Predictor.result$ 2;
  $node$ 3 $\leftarrow$ $Predictor.result$ 3;
  for 3 $node$ of $DPL$ tail do
      add $node\ i$ to $DPL$ tail;
  end for
  end function

  function MoveToNVM($node$ n)
  remove n from DPL;
  if n in $NPL$ then
      if $node.dirty$ is $true$ then
          write back;
      end if
  else                                                    ▷ page fault page
      add $n$ to $NPL$ head;
      write back;
  end if
  end function
```

Table 1 GEM5 simulation configuration

Parameter	Value
CPU	AtomicSimpleCPU
L1 data cache	32 KB (8-way, 64-byte line)
L1 instruction cache	32 KB (8-way, 64-byte line)
L2 cache	512 KB (8-way, 64-byte line)
L3 cache	4 MB (16-way, 64-byte line)

3 Evaluation and Analysis

3.1 Trace Collection

We use the SPEC CPU2006 benchmarks with the simulator GEM5, and totally 10 benchmarks are run for 100 million instructions respectively to collect detailed traces of memory access patterns. The simulation is set up with a three-level cache architecture, which is common in many modern CPUs. AtomicSimpleCPU is used in GEM5 to prioritize simulation speed. The configuration details are provided in Table 1.

To simulate memory shortage situations, which are common in real-world applications, the memory capacity in the simulation can be artificially limited. The collected traces are summarized in Table 2, which includes details such as the number of memory accesses, read/write operations, and other relevant metrics. Some benchmarks exhibiting write-operations absence to memory is caused by their memory footprint not surpassing the dimensions of the L3 cache. For benchmarks with similar memory footprints, the request count can serve as an indicator of locality. Fewer requests suggest higher locality, in which case the benchmark accesses a smaller set of memory addresses more frequently. The collected traces likely reflect a variety of localities based on different memory footprint sizes, which is important for training and evaluating the TransMigrator system's ability to predict effectively.

Table 2 Trace characteristics. *"Ifetch" means instruction fetch

Benchmark	Footprint	Request	Read	Write	Ifetch
bzip2	19.4648MB	103309	87148	15509	652
gcc	11.3885MB	157223	107483	41787	7953
calculix	5.24469MB	31889	28517	0	3372
cactusADM	25.5783MB	1052375	675028	373032	4315
sjeng	175.43MB	16536109	8299780	8234681	1648
hmmer	1.65619MB	6466	5143	0	1323
mcf	5.85046MB	98296	81729	15962	605
soplex	3.35858MB	19983	16628	0	3355
omnetpp	3.88507MB	12771	9834	0	2937
povray	5.37494MB	50785	46121	1	4663

Table 3 Network training configuration

Parameter	Value
encoder layers	2
decoder layers	2
number of heads	2
model dimension	64
sequence length	30
token length	8
learning rate	0.001
batch size	2048
dropout	0.0
seed	43
loss	cross entropy
optimizer	adam

3.2 Network Training

Dataset The training and validation dataset is a mixture of all collected traces. Traces are divided into smaller sequences of length $2\times L_{seq}$. Each small sequence is split into two halves; the first half is used as input, and the latter half produces real labels. The label is determined by the most frequently appearing page number in the latter half. A stride of $0.5\times L_{seq}$ is used during splitting to generate more sequences. Sequences are randomly divided into a training set and a validation set in a 7:3 ratio. Due to severe imbalance in the number of sequences per benchmark, a weighted sampler is employed. The weight for sequences from benchmark i is $\frac{1}{n_i}$, where n_i is the number of sequences for that benchmark. This approach gives smaller traces a higher chance of being included in a training batch.

Network configuration A cosine learning rate scheduler with a warm-up stage is used for stable convergence. The learning rate factor is calculated using an equation that considers the iteration number in batches (i), warm-up steps (ω), and the maximum iteration (m). Table 3 would detail the specific configuration parameters of the network.

$$lrfactor = \begin{cases} 0.5(1 + cos(\frac{i}{m}\pi)) & i \geq \omega \\ \frac{i}{\omega}, & i < \omega \end{cases}, \tag{1}$$

Results Traces are split into sequences of length L, and the network predicts whether the predicted page is the most frequent in the next sequence. As shown in Table 4, the network achieves an average prediction accuracy of 0.7245 in 10 benchmarks. Given the scale of the network, this result is considered remarkable. Despite the high accuracy, the network alone may not be sufficient for the entire task, especially for benchmarks with lower accuracy. To compensate for potential inaccuracy, a classic threshold method is used in conjunction with the neural network.

Table 4 Network prediction accuracy

Benchmark	Accuracy
bzip2	0.648
cactusADM	0.401
calculix	0.7292
gcc	0.7638
hmmer	0.7765
mcf	0.8092
omnetpp	0.7485
povray	0.7643
sjeng	0.8000
soplex	0.8047

Table 5 Network prediction accuracy

Attributes	threshold	lifetime
AC-CLOCK	–	–
THMigrator	32	256
VC-HMM	32	256

3.3 Migration Simulation

A simulator is created to assess different page migration strategies. The neural network's predictions are pre-computed and stored in individual files for each benchmark, from which predicted page numbers are read as needed.

Configurations AC-CLOCK [13], THMigrator [19], and VC-HMM [22] are chosen for comparison. AC-CLOCK has no hyperparameters. THMigrator, VC-HMM, and the proposed design share two parameters: threshold and lifetime, both set to 32 and 256, respectively. VC-HMM utilizes a small DRAM as a direct mapping victim cache between DRAM and NVM, a feature also considered in the simulation setup. The parameters for the simulator, as shown in Table 5, are carefully chosen to reflect real-world usage.

To simulate intensive memory usage, the NVM size for each benchmark is set to approximate the memory footprint. The sizes of DRAM, NVM, and victim cache are set in a ratio of 1:8:$\frac{1}{16}$, as detailed in Table 6 for each benchmark.

Evaluations Total access time and total energy consumption are defined by Eqs. 2 and 3, which are derived from APMigrate with modifications.

When ascertaining the total access time, we disregard certain page migration costs. This is feasible because the migration of pages can occur in the intervals between memory access events. Absent this oversight, the efficiency of the system would be adversely affected by each page migration due to the substantial overhead associated with it, which is significantly greater than that of routine read and write operations within a cache line. On the other hand, in energy consumption evaluations, we do not follow such disregard, as the energy consumption is inherently observable and cannot be circumvented.

Table 6 Benchmarks configuration. * “VC” means victim cache used in VC-HMM

Benchmark	NVMsize	DRAMsize	VCsize
bzip2	20.0MB	2.5MB	128.0KB
gcc	12.0MB	1.5MB	64.0KB
calculix	6.0MB	768.0KB	32.0KB
cactusADM	26.0MB	3.25MB	128.0KB
sjeng	176.0MB	22.0MB	1.0MB
hmmer	2.0MB	256.0KB	16.0KB
mcf	6.0MB	768.0KB	32.0KB
soplex	4.0MB	512.0KB	32.0KB
omnetpp	4.0MB	512.0KB	32.0KB
povray	6.0MB	768.0KB	32.0KB

Variables used in Eqs. 2 and 3 are explained in Table 7. The migration time is set to 380 times the DRAM access time according to APMigrate. Other variable values are sourced from [8] and listed in Table 8.

Table 7 Variables in computation

Symbol	Description
n_{dr}	DRAM read count
n_{dw}	DRAM write count
n_{nr}	NVM read count
n_{nw}	NVM write count
n_{mig}	migration count
t_{dr}	DRAM read latency
t_{dw}	DRAM write latency
t_{nr}	NVM read latency
t_{nw}	NVM write latency
t_{mig}	migration latency
$size_d$	DRAM size
$data_{dr}$	total data of DRAM read
$data_{dw}$	total data of DRAM write
$data_{nr}$	total data of NVM read
$data_{nw}$	total data of NVM write
e_{dr}	DRAM read energy
e_{dw}	DRAM write energy
e_{nr}	NVM read energy
e_{nw}	NVM write energy
p_d	DRAM static power
p_n	NVM static power
$size_n$	NVM size

Table 8 Memory characteristics

Attributes	DRAM	NVM
latency read (ns)	15	50
latency write (ns)	15	150
energy read (mJ/GB)	12.1857	15.62
energy write (mJ/GB)	3.0469	150
energy static (mW/GB)	100	1

$$
\begin{aligned}
t_{total} = {} & n_{dr} \times t_{dr} + n_{dw} \times t_{dw} \\
& + n_{nr} \times t_{nr} + n_{nw} \times t_{nw} \\
& + n_{mig} \times t_{mig}
\end{aligned} \tag{2}
$$

$$
\begin{aligned}
e_{total} = {} & data_{dr} \times e_{dr} + data_{dw} \times e_{dw} \\
& + data_{nr} \times e_{nr} + data_{nw} \times e_{nw} \\
& + (p_d \times size_d + p_n \times size_n) \times t_{total}
\end{aligned} \tag{3}
$$

3.4 Access Time

As demonstrated in Fig. 5, TransMigrator, significantly reduces the average access time compared to THMigrator, VC-HMM, and AC-CLOCK by 38.72%, 26.27%, and 5.41%, respectively, with data normalized by dividing by the mean for each benchmark to ensure consistent scales. This indicates that TransMigrator is more efficient in managing memory access times.

THMigrator is described as conservative in the experiment setting, making fewer migrations than other approaches. This introduces very low energy consumption but leaves many pages in NVM, leading to increased access times. However, on benchmarks gcc and sjeng, THMigrator performs better with more page migrations. Due to the limited capacity of DRAM, migrating certain pages is necessary for better performance. Too few migrations can undermine the system's performance. VC-HMM, which updates the threshold based on benefits, is highlighted as having a potential flaw. There is often a long delay between the start of migration and the adjustment of the threshold. When low beneficial migrations are detected, the threshold can be pushed to an extreme value, preventing further migrations and making it difficult for the threshold to return to a normal range. Despite the potential issue with threshold adjustment, VC-HMM generally performs better than THMigrator in most cases, except for the risk of falling into the trap of extreme threshold values. AC-CLOCK is noted for achieving relatively low access times compared to THMigrator and VC-HMM and is considered robust across all benchmarks. TransMigrator achieves lower access times than AC-CLOCK in most benchmarks, except for bzip2, where

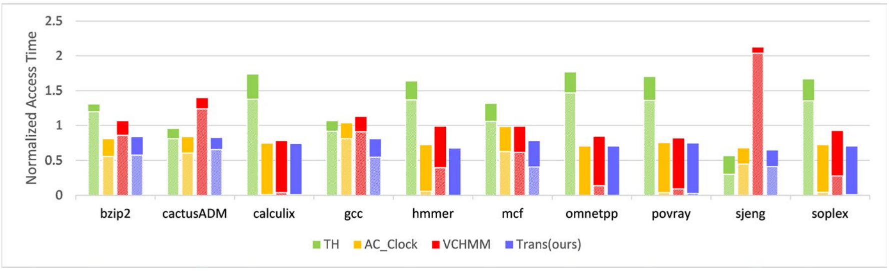


Fig. 5 Normalized access time. The complete bar represents total access time. The shadow portion indicates NVM access time

it has more NVM writes. The design does not distinguish between read and write accesses, which may lead to pages with many write accesses not being migrated based on access count. TransMigrator demonstrates robust and good performance in most benchmarks. It achieves low access time in the benchmark cactusADM, even with a prediction accuracy of only 40.10%. This suggests that the combination of the predictor and the fallback method is effective and robust. The effectiveness of the co-work between the predictor and the fallback method in TransMigrator is emphasized, contributing to its overall performance and robustness.

3.5 Energy Consumption

As depicted in Fig. 6, the proposed method, TransMigrator, saves the average total energy consumption by 25.04% compared to VC-HMM but consumes more energy than AC-CLOCK by 10.22%. TransMigrator consumes 2.5 times the energy of THMigrator, which is more energy-efficient due to its conservative approach and fewer migrations. However, THMigrator's lower energy consumption comes at the cost of slower access times. Non-Volatile Memory (NVM) is the dominant factor in energy consumption. Despite its low static power, NVM has a significant overhead for read and write operations compared to DRAM, especially during page migrations. The more page migrations that occur, the higher the energy consumption, as illustrated in Fig. 7. More page migration does not necessarily reduce access time. For example, on the hmmer and omnetpp benchmarks, VC-HMM performs more migrations than TransMigrator but has a higher access time. VC-HMM has the lowest number of migrations but the highest energy consumption. This is because regular accesses to NVM become the primary source of energy consumption when the number of migrations is reduced. AC-CLOCK and TransMigrator achieve a better balance between energy consumption and access time compared to the other two approaches. TransMigrator is noted for its efficiency, reducing energy consumption compared to VC-HMM while still providing faster access times than THMigrator.

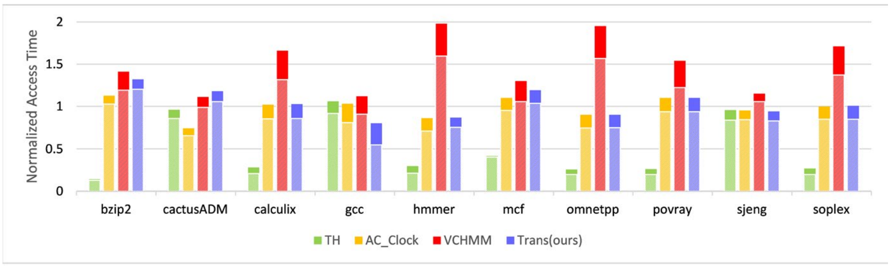


Fig. 6 Normalized energy consumption. The complete bar represents total energy consumption. The shadow part indicates NVM consumption

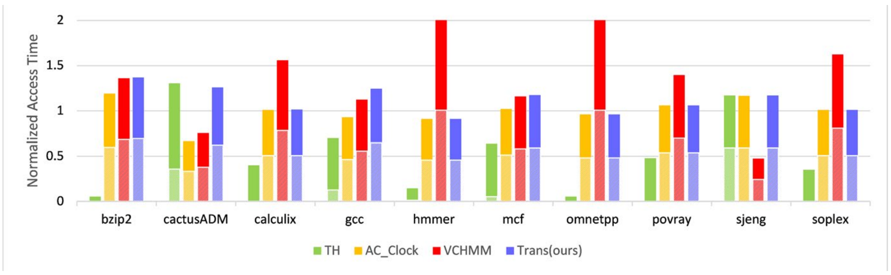


Fig. 7 Normalized migration count.The complete bar represents the sum count of migration from NVM to DRAM and writing back. The shadow portion indicates the NVM actual write back count, with clean pages excluded

3.6 Network Overhead

The prediction part (neural network) and the migration part of the system operate in parallel. This means they move independently, with the migration part always performing its routine tasks. The inference overhead of the neural network can be partially hidden due to the parallel nature of the system. This suggests that the time taken for the network to make predictions does not significantly impact the overall system performance because other processes continue to run concurrently. The system is designed to work effectively with or without the network part. The neural network enhances the system's capabilities but is not essential for its basic operation. The latency of the system is influenced more by the prediction accuracy of the network than by its inference speed. This implies that even if the network takes some time to make predictions, as long as those predictions are accurate, the system's performance in terms of latency will be acceptable. The network only needs to commit an inference every multiple of memory accesses. This means that the frequency of predictions is not as critical as the accuracy of those predictions. Given the inference frequency and the ability to apply various techniques to improve the network's performance, there is room to use more sophisticated models. These models could potentially offer better prediction accuracy without causing a noticeable increase in latency. There is a trade-off between using a more complex model for better accuracy and maintaining low latency. However, due to the system's design and operation, it is possible to leverage more complex models without significantly impacting latency.

4 Related Work

DPQ [23] presents a promising approach for deploying deep neural networks on resource-constrained edge devices, for pruning and compressing deep neural network models. The technique offers a balance between model size and accuracy, which is essential for practical applications in computility-limited deployment scenarios. SPARK [15] utilizes a variable-length scheme to optimize both the algorithmic and architectural aspects for local parameter training. It represents an alternative approach to model compression that addresses the challenges of deploying large DNNs on constrained hardware. The work of Songwen Pei et al. [21] from the aspect of pruning optimizing, proposes a novel filter pruning algorithm, named DRP. With a co-designed method called CBS, the solution performs better in pruning networks of huge sizes, and makes it easier to deploy large-scale models on devices with limited resources.

The work Seok et al. [25] focuses on moving write-bounded pages to DRAM and read-bounded pages to NVM, using a moving average weight updated by access type and thresholds determined by trials. Clock-DWF [14] is Similar to [25], which aims to migrate write operations to DRAM but uses a CLOCK algorithm for page selection and eviction based on dirty bits. Adaptive-Classification CLOCK [13] is an improvement over Clock-DWF, the approach of which places all fault pages in DRAM and enforces a stricter migration policy.

UIMigrate [26] calculates page hotness with decay factors and adjusts thresholds based on migration benefits. APMigrate [27] builds on UIMigrate, writing back only changed parts of a page back to NVM using a bitmap. THMigrator [20] utilizes two-way hash chain lists for accelerated page lookup. VC-HMM [22] Introduces a victim cache between DRAM and NVM to reduce NVM write operations. AMP [12] Employs multiple strategies (LRU, LFU, random) and switches between them based on page access ratios and LRU hit ratios. Adavally et al. [1] Adjusts migration thresholds by monitoring migration counts and average access counts, using a small mapping table for page repositioning. OAM [16] Operates at the program object level, combining access time and energy consumption, and injects allocation/migration instructions via the compiler. Kleio [10] Selects important pages and trains a separate recurrent neural network for each, using predicted access counts for migration decisions. DeepSwapper [4] Uses LSTMs to predict future address differences and migrates frequently written pages to NVM, unlike TransMigrator, which treats read and write equally. Doudali et al. [11] Maps page accesses to image pixels, marking areas for important pages based on access patterns. TransFetch [29] is Designed for prefetching data to the cache rather than page migration. It uses a similar input splitting method and transformer architecture as TransMigrator but differs in its approach to embeddings and output labeling. Long et al. [17] proposes prefetching an additional page on page faults for both CPUs and GPUs, modeling the problem as a classification task using transformer encoders for prediction.

5 Conclusion

The article proposes TransMigrator, a novel approach to page migration in hybrid memory systems that combine DRAM and NVM. The core of TransMigrator is an end-to-end neural network with a transformer backbone, which is used to predict future hot pages based on previous memory access patterns. The network achieves an average prediction accuracy of 0.7245 across different benchmarks, with an estimated model parameter size of 0.804 MB. TransMigrator combines the transformer-based predictor with a fixed threshold migration approach. The latter serves as a fallback mechanism when the predictor's accuracy is low.

Experiments demonstrate that TransMigrator significantly outperforms other migration strategies like THMigrator, VC-HMM, and AC-CLOCK, reducing average access time by 38.72%, 26.27%, and 5.41%, respectively. The dual approach of prediction and fallback ensures accurate migrations and maintains low latency, even in cases of mispredictions.

Limitations: The simulator used for evaluation lacks timing information, which means some migration overheads are not fully accounted for. The neural network does not differentiate between read and write operations. The fallback method is considered simplistic and may not be optimal. These limitations suggest that the neural network could be integrated with other page migration strategies. Improvements to the network design, and fallback mechanisms are identified as areas for future research.

Acknowledgements We would like to thank the anonymous reviewers for their invaluable comments. This work was partially funded by the National Natural Science Foundation of China under Grant 61975124, Shanghai Natural Science Foundation (20ZR1438500), State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No.CARCHA202111, and Engineering Research Center of Software/Hardware Co-design Technology and Application, Ministry of Education East China Normal University under Grant No.OP202202. Any opinions, findings and conclusions expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

Author contributions Songwen Pei propose the original migrator mechnism, revise the manuscript, response comments, etc. Wei Qin discuss on the structure of TransMigrator, edit some figures, and rewirte it. Jinan Li edit the orginal manuscript, and do experiments and collect data. Junhao Tan edit and format part of the new version of manuscript. Jie Tang discuss the main contributions, and fix some typos. Jean-Luc Gaudiot suggests on the structure of the paper, polish some expression.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Adavally, S., Islam, M., Kavi, K.: Dynamically adapting page migration policies based on applications' memory access behaviors. J. Emerg. Technol. Comput. Syst. (2021). https://doi.org/10.1145/3444750
2. Aryana, K., Gaskins, J.T., Nag, J., Stewart, D.A., Bai, Z., Mukhopadhyay, S., Read, J.C., Olson, D.H., Hoglund, E.R., Howe, J.M., Giri, A., Grobis, M.K., Hopkins, P.E.: Interface controlled thermal resistances of ultra-thin chalcogenide-based phase change memory devices. Nat. Commun. **12**, 774 (2021). https://doi.org/10.1038/s41467-020-20661-8
3. Beigi, M. V., Pourshirazi, B., Memik, G., Zhu, Z.: Deepswapper: a deep learning based page swap management scheme for hybrid memory systems. In: Sarkar, V., Kim, H. (Eds.), PACT '20: International conference on parallel architectures and compilation techniques, virtual event, GA, USA, October 3-7, 2020 pp. 353–354. ACM (2020a). https://doi.org/10.1145/3410463.3414672
4. Beigi, M. V., Pourshirazi, B., Memik, G., Zhu, Z.: Deepswapper: a deep learning based page swap management scheme for hybrid memory systems. In: Proceedings of the ACM international

conference on parallel architectures and compilation techniques, **10**(1145/3410463), 3414672 (2020)

5. Burr, G.W., Brightsky, M.J., Sebastian, A., Cheng, H.-Y., Wu, J.-Y., Kim, S., Sosa, N.E., Papandreou, N., Lung, H.-L., Pozidis, H., Eleftheriou, E., Lam, C.H.: Recent progress in phase-change memory technology. IEEE J. Emerg. Sel. Top. Circuits Syst. **6**, 146–162 (2016). https://doi.org/10.1109/JETCAS.2016.2547718
6. Cappelletti, P.: Non volatile memory evolution and revolution. In: 2015 IEEE International Electron Devices Meeting (IEDM) pp. 10.1.1–10.1.4 (2015). https://doi.org/10.1109/IEDM.2015.7409666
7. Chen, A.: A review of emerging non-volatile memory (NVM) technologies and applications. Solid-State Electron. **125**, 25–38 (2016). https://doi.org/10.1016/j.sse.2016.07.006
8. Chen, T.-Y., Chang, Y.-H., Chen, S.-H., Kuo, C.-C., Yang, M.-C., Wei, H.-W., Shih, W.-K.: Enabling write-reduction strategy for journaling file systems over byte-addressable nvram. In: 2017 54th ACM/EDAC/IEEE Design automation conference (DAC) pp. 1–6 (2017). 10.1145/3061639.3062236
9. Dayarathna, M., Wen, Y., Fan, R.: Data center energy consumption modeling: a survey. IEEE Commun. Surv. Tutor. **18**, 732–794 (2016). https://doi.org/10.1109/COMST.2015.2481183
10. Doudali, T.D., Blagodurov, S., Vishnu, A., Gurumurthi, S., Gavrilovska, A.: Kleio: A hybrid memory page scheduler with machine intelligence. In: Proceedings of the 28th International symposium on high-performance parallel and distributed computing (2019). https://api.semanticscholar.org/CorpusID:195325868
11. Doudali, T.D., Gavrilovska, A.: Toward computer vision-based machine intelligent hybrid memory management. In: Proceedings of the international symposium on memory systems MEMSYS '21. New York, NY, USA: Association for Computing Machinery (2022). 10.1145/3488423.3519325
12. Heo, T., Wang, Y., Cui, W., Huh, J., Zhang, L.: Adaptive page migration policy with huge pages in tiered memory systems. IEEE Trans. Comput. **71**, 53–68 (2022). https://doi.org/10.1109/TC.2020.3036686
13. Kim, S., Hwang, S.-H., Kwak, J.W.: Adaptive-classification clock: Page replacement policy based on read/write access pattern for hybrid dram and PCM main memory. Microprocess. Microsyst. **57**, 65–75 (2018). https://doi.org/10.1016/j.micpro.2018.01.003
14. Lee, S., Bahn, H., Noh, S.H.: Clock-dwf: a write-history-aware page replacement algorithm for hybrid PCM and dram memory architectures. IEEE Trans. Comput. **63**, 2187–2200 (2014). https://doi.org/10.1109/TC.2013.98
15. Liu, F., Yang, N., Li, H., Wang, Z., Song, Z., Pei, S., Jiang, L.: Spark: Scalable and precision-aware acceleration of neural networks via efficient encoding. In: 2024 IEEE International symposium on high-performance computer architecture (HPCA) pp. 1029–1042 (2024). 10.1109/HPCA57654.2024.00082
16. Liu, H., Liu, R., Liao, X., Jin, H., He, B., Zhang, Y.: Object-level memory allocation and migration in hybrid memory systems. IEEE Trans. Comput. **69**, 1401–1413 (2020). https://doi.org/10.1109/TC.2020.2973134
17. Long, X., Gong, X., Zhang, B., Zhou, H.: Deep learning based data prefetching in CPU-GPU unified virtual memory. J. Parallel Distrib. Comput. **174**, 19–31 (2023). https://doi.org/10.1016/j.jpdc.2022.12.004
18. Mittal, S., Vetter, J.S.: A survey of software techniques for using non-volatile memories for storage and main memory systems. IEEE Trans. Parallel Distrib. Syst. **27**, 1537–1550 (2016). https://doi.org/10.1109/TPDS.2015.2442980
19. Pei, S., Ji, Y., Shen, T., Liu, H.: Migration mechanism of heterogeneous memory pages using a two-way hash chain list. SCI. SIN. Inf. **49**(9), 1138–1158 (2019)
20. Pei, S., Li, J., Qian, Y., Tang, J., Gaudiot, J.-L.: Transmigrator: a transformer-based predictive page migration mechanism for heterogeneous memory. In: Liu, S., Wei, X. (eds.) Network and Parallel Computing, pp. 180–191. Springer, Cham (2022)
21. Pei, S., Luo, J., Liang, S., Ding, H., Ye, X., Chen, M.: Carbon emissions reduction of neural network by discrete rank pruning. CCF Trans. High Perform. Comput. **5**, 334–346 (2023)
22. Pei, S., Qian, Y., Ye, X., Liu, H., Kong, L.: Dram-based victim cache for page migration mechanism on heterogeneous main memory. J. Comput. Res. Develop. **59**(3), 568–581 (2022)
23. Pei, S., Wang, J., Zhang, B., Qin, W., Xue, H., Ye, X., Chen, M.: DPQ: dynamic pseudo-mean mixed-precision quantization for pruned neural network. Mach. Learn. **113**, 4099–4112 (2024). https://doi.org/10.1007/s10994-023-06453-3

24. Raoux, S., Xiong, F., Wuttig, M., Pop, E.: Phase change materials and phase change memory. MRS Bull. **39**, 703–710 (2014)
25. Seok, H., Park, Y., Park, K.-W., Park, K.H.: Efficient page caching algorithm with prediction and migration for a hybrid main memory. SIGAPP Appl. Comput. Rev. **11**, 38–48 (2011). https://doi.org/10.1145/2107756.2107760
26. Tan, Y., Wang, B., Yan, Z., Deng, Q., Chen, X., Liu, D.: Uimigrate: adaptive data migration for hybrid non-volatile memory systems. In: 2019 Design, automation & test in Europe conference & exhibition (DATE) pp. 860–865 (2019). 10.23919/DATE.2019.8715118
27. Tan, Y., Wang, B., Yan, Z., Srisa-an, W., Chen, X., Liu, D.: Apmigration: improving performance of hybrid memory performance via an adaptive page migration method. IEEE Trans. Parallel Distrib. Syst **31**, 266–278 (2020). https://doi.org/10.1109/TPDS.2019.2933521
28. Vetter, J.S., Mittal, S.: Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. Comput. Sci. Eng. **17**, 73–82 (2015). https://doi.org/10.1109/MCSE.2015.4
29. Zhang, P., Srivastava, A., Nori, A. V., Kannan, R., Prasanna, V. K.: Fine-grained address segmentation for attention-based variable-degree prefetching. In: Proceedings of the 19th ACM international conference on computing frontiers CF '22 pp. 103-112. New York, NY, USA: Association for Computing Machinery (2022). 10.1145/3528416.3530236