2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems

Energy-Efficient Data Caching Framework for Spark in Hybrid DRAM/NVM Memory Architectures

Bo Wang
School of Computer Science and Technology
Beijing Institute of Technology University
Beijing, China
wangsit@netease.com

Jie Tang
School of Computer Science and Engineering
South China University of Technology
Guangzhou, China
cstangjie@scut.edu.cn

Rui Zhang
College of Education Science
Yan'an University
Yan'an, China
zr@yau.edu.cn

Wei Ding
College of Information Science and Engineering
Henan University of Technology
Zhengzhou, China
dingwei@haut.edu.cn

Shaoshan Liu
PerceptIn
shaoshan.liu@perceptin.io

Deyu Qi
School of Computer Science and Engineering
South China University of Technology
Guangzhou, China
csqideyu@scut.edu.cn

***Abstract*—In Spark, a typical in-memory big data computing framework, an overwhelming majority of memory is used for caching data. Among those cached data, inactive data and suspension data account for a large portion during the execution. These data remain in memory until they are expelled or accessed again. During the period, DRAM needs to consume a lot of refresh energy to maintain these low profit data. Such a great energy waste can be terminated if we use NVM as alternation. Meanwhile, NVM is smaller cell-sized that it provides more in-memory room for caching data instead of disk access in DRAM setting. However, NVM can not completely take the place of DRAM due to its superiority in terms of access latency and endurance. So, hybrid DRAM/NVM memory architectures turns to be the optimal solution and have a promising prospect to solve the memory capacity and energy consumption dilemmas for in-memory big data computing systems. With this observation, in this paper, we propose a data caching framework for Spark in hybrid DRAM/NVM memory configuration. By identifying the data access behaviors with active factor and active stage distance, cache data with higher local I/O activity is prioritized cached in DRAM, while cache data with lower activity is placed into NVM. The data migration strategy dynamically moves the cold data from DRAM into NVM to save static energy consumption. The result shows that the proposed framework can effectively reduce energy consumption about 73.2% and improve latency performance by up to 20.9%.**

Keywords— Big data, Data placement, Energy consumption, Non-Volatile Memory;

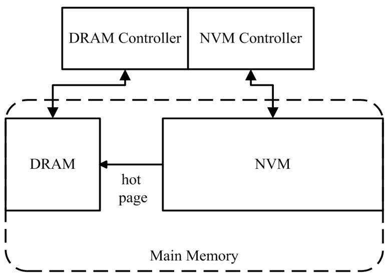
I. Introduction

Nowadays, internet companies have built clusters with thousands nodes to process big data. These clusters consumed a good deal of energy. The energy costs make up about 42% of the data centers' operating costs and it growing at over 15% annually in US [1]. For in-memory computing systems, which are prevalent in both industry and academia for big data computing [2], memory accounts for up to 40% of total energy consumption [3]. Additionally, memory capacity and energy consumption keeps increasing with great data process demand due to data explosion. Subject to physical limitations, it is difficult to further scale DRAM (Dynamic Random Access Memory) in terms of capacity now. As a result, existing in-memory big data computing systems, which use single DRAM memory architectures, are in trouble with energy consumption and capacity bottlenecks [4].

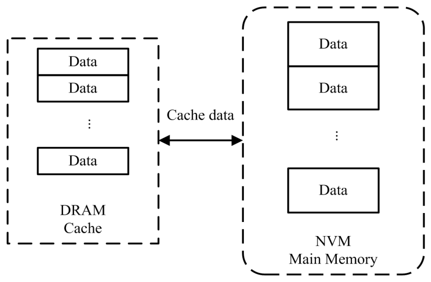
TABLE I. Performance parameters between NVM and DRAM [19]

Feature	*DRAM*	*PCM*	*STT-RAM*
Cell Size(F^2)	6~8	4~8	16~60
Read Latency(ns)	<10	10~100	2~20
Write Latency(ns)	<10	20~120	5~35
Write power($nj \cdot b^{-1}$)	~0.1	<1	1.6~5
Leak power	High	Low	Low
Nonvolatility	No	Yes	Yes
Endurance(times)	N/A	10^8~10^{12}	10^{12}~10^{15}

Emerging NVM (Non-Volatile Memory) technologies, which has high storage density and lower bit cost (Cost-Per-Bit) and zero static energy consumption (as shown in Table I), are being explored as potential alternatives to DRAM in in-memory big data computing systems [5]. For example, under the same on-chip area, the capacity of PCM (Phase-Change RAM), one type of NVM, is four times that of DRAM. On the flip side, NVM has higher access latency and write power consumption. Either DRAM or NVM can not meet both fast access and low power consumption simultaneously. Thus hybrid DRAM/NVM memory architectures are more promising for building high-performance, large-scale, and energy efficient main memory systems [6,7]. Currently, there are flat-addressable hybrid memory architecture [6] and hierarchical hybrid memory architecture [7] two typical hybrid memory architectures, as shown in Fig.1. Compared to the hierarchical hybrid memory architecture in which high metadata-managed overhead limits the maximum capacity of NVM, flat-addressable hybrid memory architecture has better scalability.



a. Flat addressable hybrid memory architecture



b. Hierarchical hybrid memory architecture

Fig.1. Typical hybrid DRAM/NVM memory architecures.

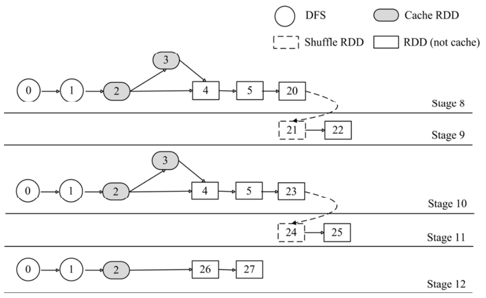


Fig.2. Segment DAG in KMeans.

Existing researches on hybrid DRAM/NVM memory architectures compensate NVM and DRAM access performance gap through hot data migration technologies. For example, some hot page migration strategies [8-10] move hot NVM pages, on which have frequent reading and writing operations, to DRAM, while remaining cold NVM pages, on which have seldom I/O operations, in NVM. However, Spark [11], an open-source in-memory big data computing framework from Apache, does not support distinguishing hot/cold data when they are stored into memory. Under the existing mechanism, the data is randomly assigned to DRAM or NVM. Moreover, the bases for determining how hot/cold a page is in these strategies are the memory access history of the page. The history-based ways are agnostic to the characteristics of big data applications. For big data applications, the access pattern of cache data is usually pre-aware. For example, Spark represents dependencies among data in the form of Directed Acyclic Graph (DAG) which is created by DAGScheduler before the application execution. DAG provides rich semantics of underling data access patterns, such as which cache data will be accessed in which future. We will have chance to use the hints in DAG to indicate what and how to migrate data in hybrid memory.

In Spark, its java heap is divided into execution memory and storage memory, which respectively corresponds to execution data and cache data. Some studies distinguish between hot and cold data using garbage collectors. For example, Wang et al. [13] map the young generation and old generation of Java GC (Garbage Collection) to DRAM and NVM, respectively, and the Java GC is responsible for migrating the data in execution memory according to some rules. While data caching in storage memory is left to the programmer to distinguish manually. Obviously, they believe that the hot and cold nature of cache data is static. However, the activity of cache data is changing during execution [2,13]. [14] further divides the cached data into active data, inactive data and suspension data. For cache data, DRAM needs to periodically refresh the circuits of the rows and columns in which the data is located to ensure the validity of the data, and consumes plenty of static energy during this period, while NVM does not need to retain them through refreshing, it only spends nearly zero static energy to do so. Further, inactive data and suspension data, which will be no longer accessed or at least not be accessed recently, occupy a significant proportion in cache data. Thus we can conclude that DRAM is the right place to reserve active data for its benefits of fast access and gain of system performance. NVM is an optimal choice for inactive data and suspension data to reduce energy consumption. The hybrid composition of these two storage gives the chance for both high performance and low power cost.

With this observation, in this paper, we first analyzed the energy consumption of cache data in Spark with single DRAM memory architecture through experiments. Then, we proposed a data caching framework for Spark in hybrid DRAM/NVM memory. To minimum storage memory energy consumption, our proposed framework places and migrates cache data between DRAM and NVM by their local active factor and active stage distance, correspondingly. According to the framework, cache data with high local active factor is preferentially stored in DRAM and cache data with low local active factor has priority to be stored in NVM. The framework also can migrate cached data from DRAM to NVM or vice versa.

The remainder of the paper is organized as follows. Sec. II demonstrates the background and energy consumption in existing single DRAM memory in Spark through experiments. The cache data placement framework in hybrid memory is elaborated in Sec. III. Evaluation results are reported in Sec. IV. We survey related work in Sec. V and conclude the paper in Sec. VI.

II. BACKGROUND AND OBSERVATION

A. Non-Volatile Memory

NVM technologies, such as Phase-Change RAM, Spin-Transfer Torque RAM (STT-RAM), and Resistive RAM

(RRAM), have some clear different characteristics from DRAM, as shown in Table I. Among them, PCM which has less feature size and lower leak power than DRAM, is the most promising alternative to DRAM in industry. It allows the computer to get large-scale on-chip memory and low energy consumption on keeping data in it. On the other hand, PCM is seriously inferior to DRAM in terms of access latency and write power consumption. Although it has a read latency close to DRAM, the write latency is several times that of DRAM. In terms of power consumption, there is a serious imbalance in the reading and writing power consumption of PCM. Writing the same data to the PCM takes more power than reading the data. Therefore, when placing data in a hybrid memory, it is necessary to consider not only the frequency of access but also the type of access.

As shown in Fig.1, the hybrid memory systems composed of NVM and DRAM mainly have hierarchical and flat-addressable hybrid memory architecture two kinds of architectures. In hierarchical hybrid memory systems, DRAM is used as a cache of NVM. This architecture can benefit from DRAM in high performance and endurance while scale main memory size with NVM [9]. Plenty of metadata are cached in DRAM cache would lead to an increase in latency and a decrease in system performance. While the flat-addressable hybrid architecture is immune to this problem. Under this architecture, DRAM and NVM are uniformly addressed. Hot data and cold data are placed in DRAM and NVM, respectively. It can provide greater physical memory for the system, which facilitates in-memory big data computing systems. Therefore, we use the flat-addressable hybrid DRAM/NVM architecture in our framework. Moreover, reasonable data migration between the two heterogeneous memories is one of the problems we need to balance.

B. Directed Acyclic Graph and Caching mechanism in Spark

Spark divides the memory space into storage memory, which usually takes up a significant proportion of the java heap, and execution memory two regions depending on using function. Among them, the storage memory stores the cache Resilient Distributed Datasets (RDDs) blocks, which are manually specified by the programmer during the encoding period, and the execution memory stores spill data made by shuffle operations. The data in storage memory and execution memory are consistent with the characteristics of data path and control path, respectively. They also follow the epochal hypothesis and the generational hypothesis, respectively [12]. For some data in storage memory may reside in memory through the whole application execution, if necessary, and contribute major of memory consumption. As a result, the size of free space in the storage memory even java heap is so short that garbage collection operations become frequent and operating costs increase. For the most data in execution memory may become unreachable quickly after allocated and have short life spans. Java GC is well qualified for the management of these data.

Spark organizes workflow of an application as a DAG. As shown in Fig. 2, there are three kinds of RDDs: shuffle RDDs, which store in distributed file system during shuffle operations; cache RDDs, which store in storage memory by

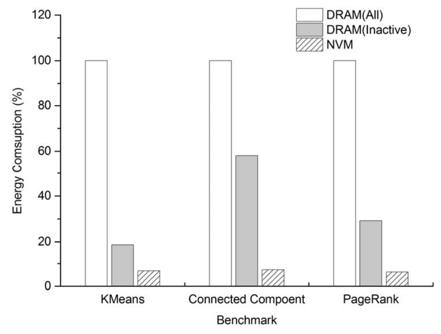


Fig. 3. Comparison of total energy, inactive data energy and nvm energy.

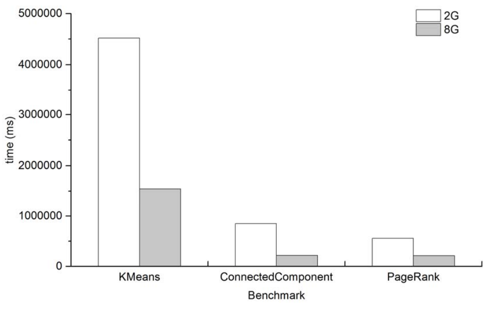


Fig. 4. Execution time in different memory sizes.

BlockManager; and un-cache RDDs. They are connected by directed edges, which indicate operations, for instance *map(), union()* and *reduce()*, on them. Before the application is executed, Spark first analyzes and divides the DAG of the application through DAGScheduler. After partition, RDDs are separated into different stages through shuffle dependencies, a many to many relationship. Subsequently, Spark executes tasks in one stage by one stage. Tasks in a new stage would not be executed until all the tasks in the previous stage have been finished. Usually, a new stage starts from a shuffle RDD or DFS file.

C. Energy consumption in DRAM Memory

To illustrate the memory energy consumption of the cache data in Spark, we analyzed the memory energy consumption, which is obtained through the NVMain [15] simulator, of three typical big data applications that are generated by Sparkbench [16] through experiments. These applications are executed on a real Spark cluster. We recorded the storage memory access operations on each executor and summed up the energy consumption of all executors of each application. We have the following observations.

1. *Inactive data contributes a significant portion of the total energy consumption.* In Fig. 2, RDD 3 becomes inactive after stage 10 finished, and it keeps inactive in stage 11 and stage 12. For big data applications especially iterative computing application, inactive data accounts for a large portion of memory footprint for some applications, with the median percentile being 77% [2]. These data residents in DRAM and consumes plenty of static energy. As Fig. 3 shown, it accounts for up to 58% of the total cache memory access energy in

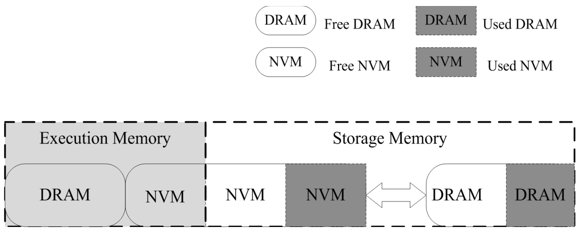


Fig. 5. Memory architecture.

Connected Component, at least 20% of the total cache memory access energy in KMeans. Each node has the similar energy consumption. Conversely, NVM consumes almost zero energy consumption on data storage. These static energy can be eliminated in NVM.

2. *The energy consumption gap between NVM and DRAM is huge during executing the same application.* As shown in Table I, NVM outperforms DRAM in power consumption, except for writing power consumption. Our experiment also verified this. Fig. 3 also shows energy consumption of three benchmarks in NVM and in DRAM, individually. Obviously, NVM has lower energy consumption, it is about 10% energy consumption of DRAM, than DRAM when storing cached data. Therefore, NVM has significant energy consumption performance without considering latency.

In addition, on the same on-chip area, NVM can store about 3 times more data than DRAM [17]. To illustrate the benefits of big memory volume, we also compared execution times for three benchmarks in two memory sizes. To be more representative, the memory size of the large memory schema is four times that of the small memory schema, which is similar to the storage density ratio of NVM and DRAM. Two memory schemas use the same data size for the same benchmark. As shown in Fig.4, the execution time of the small memory schema is 265-394% of the large memory schema. This is due to the fact that the large memory schema has less re-computing counts than the small memory schema. Although this experiment compares the execution time under DRAM, it can also reflect the performance improvement brought by NVM with larger on-chip memory capacity.

As can be seen from the above, NVM can make the system profitable in terms of energy consumption and system performance. It has lower energy consumption than DRAM. Putting a large proportion of inactive cache data into NVM can effectively reduce static energy consumption. Simultaneously, active data can obtain more DRAM memory space, which can improve system access efficiency. In view of this, we propose a hybrid DRAM/NVM architecture memory system that supports data caching in in-memory big data computing system.

III. ENERGY-EFFICIENT DATA CACHING FRAMEWORK

A. Memory Model

Figure 5 shows our memory architecture, which is similar to the architecture proposed by Wang et al. [13]. It can be found that execution memory and storage memory are mapped in both DRAM and NVM, respectively. Execution memory and storage memory can use different memory allocators. For example: Wang et al. handed over execution memory to OpenJDK. Young generation is mapped to DRAM, while old generation is mapped to NVM. The Java GC adjusts it to DRAM or NVM based on the activity of the data. The data in the storage memory is managed by the user. Here we only discuss storage memory.

Definition 1 (Available storage memory) For an executor e, its available memory consists of free storage memory and inactive storage memory. Details as follows:

$$Asm_e = m_{free}^e + \sum_{b_k \in IB} size_k \qquad (1)$$

Where, m^e_{free} is the free storage memory in executor e; IB is the set that contains all the inactive cache blocks, whose *reference counts* equal to zero [2]; $size_k$ is the size of block b_k. [18] uses a similar definition to represent usable memory space. However, we further extend the available storage memory to DRAM and NVM. The available storage memory in DRAM and NVM are denoted as Asm_e^{dram} and Asm_e^{nvm} ,respectively.

B. Cache data placement policy

Definition 2 (Local active factor) For a block b_t, its active factor is a function of *reference count* and distance between current stage and next active stage, details as follows:

$$LF_t = \gamma \times (IntraRC_t^s + InterRC_t^s) / D_s^l \qquad (2)$$

Where, s is the current stage id, $IntraRC_t^s$ indicates the number of child blocks that are derived from b_t in stage s and $InterRC_t^s$ indicates the number of child blocks that are derived from b_t in downstream stages after s [14]. l is the stage id that b_t will be visited firstly in the downstream stages. D^l_s is the distance between stage s and stage l. From formula 2, we can find that LF_t represents the weight of a cache block that is stored in DRAM. Cache blocks with large LF_t are more likely to be stored in DRAM than cache blocks with small LF_t, and are instead easy to store in NVM.

Algorithm 1 Cache data placement

```
 function: Data_place()
1.  lst.sort //lst is the cache block list in current stage
2.  dram_threshold = avg(lst.lf) // avarage lf in lst is asigned to threshold
3.  while (lst is not null) do
4.    if ($LF_t$ > = dram_threshold) && (t.size < dram_asm) then
5.      flag_t = DRAM
6.      dram_avasize -= t.size
7.      dram_threshold *=1+ incr_coef(t, dram_memlst)
8.    else
9.      if (t.size < dram_avasize) then
10.       dram_threshold *= 1- decr_coef(lst, dram_threshold)
11.     endif
12.       flag_t=NVM
13.   endif
14.   t.flag = flag_t
15.   lst.remove(t)
16. endwhile
17. return
18.endfunction
```

```
19. function: incr_coef(element, memlst)
20.   mlst = memlst + element
21.   while (mlst is not null) do
22.      sum += e.lf
23.   endwhile
24.   return element.lf-sum/mlst.count
25. endfunction
26. function: decr_coef(blocklst, threshold)
27.   while (blocklst is not null) do
28.      sum += e.lf
29.   endwhile
30.   avg = sum/blocklst.count
31.   return abs((avg -threshold) / avg)
32. endfunction
```

According to our placement strategy, when a stage begins, we traverse all cache blocks executing on the executor, and give priority to place a block with a large local active factor into DRAM. Instead, blocks with small local activity factors are prioritized into NVM. We use a threshold to distinguish the type of memory on which the block is placed. The threshold can be adjusted according to the active characteristics of the cache block which would be executed in current stage and the active state of cached blocks in memory. As shown in algorithm 1, if local active factor of a cache block reaches the dram_threshold, BlockManger will store the block into DRAM memory; On the contrary, if its local active factor is less than the dram_threshold, it will be stored to NVM.

C. Cache data migration policy

Definition 3 (Active stage distance) For a cache block b_t, its active stage distance can be expressed as a function of *reference count* in next active stage and the distance between current stage and next active stage, details as follows:

$$ASD_s^l = D_s^l \,/\, IntraRC_t^l \qquad (3)$$

Where, as definition 2, s is the current stage, l is the next stage that b_t will be accessed; $IntraRC_t^l$ indicates the number of child blocks that are derived from b_t in stage l; D_s^l is the distance between stage s and stage l. As mentioned in the previous section, the cache block is accessed at different stages. In formula 3, ASD^t_s represents the distance a block is accessed again. From formula 3, we can find that blocks with larger value of ASD^t_s should stay in NVM for reducing static energy consumption.

Algorithm 2 Cache data migration

```
   function: Data_migrate()
1.  lst.sort //lst is the cache block list in dram
2.  migrate_threshold = max(nvmlst.asd)//nvmlst is the cache blocks in nvm
3.  while (lst is not null) do
4.     if (ASD_l^t > migrate_threshold) && (t.size < nvm_avasize) then
5.       flag_m = NVM
6.       nvm_avasize -= t.size
7.       gama = incr_coef(t.size, nvm_avasize,gama)
8.       migrate_threshold *= 1+ gama
9.     else
10.       if (t.size < nvm_avasize) then
11.         gama =  decr_coef(t.size, nvm_avasize,gama)
12.         migrate_threshold *= 1- gama
13.         flag_m=DRAM
14.       endif
15.     endif
16.     t.flag = flag_m
17.     lst.remove(t)
18.  endwhile
19.  return
20. endfunction
21. function: decr_coef(size, avasize, coef)
22.    return coef -= coef*size/avasize
23. endfunction
24. function: incr_coef(size, avasize, coef)
25.    return coef += coef*(avasize-size)/avasize
26. endfunction
```

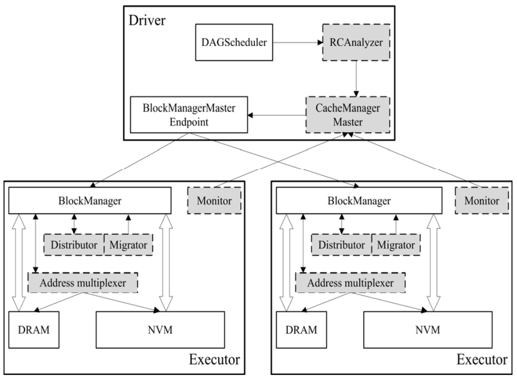


Fig. 6. Overall architecture.

TABLE II. SYSTEM CONFIGURATION OF PCM AND DRAM

Memory	*Parameters*
DRAM	1 GB: 1 channel, 1 rank, 8 banks, 65536 rows, 32 cols, Timing(tCAS-tRCD-tRP-tRAS):10-9-9-24(cycles), 19 ns read latency, 24 ns write latency
PCM	3 GB: 3 channels, 3 ranks, 8 banks, 65536 rows, 32 cols, Timing(tCAS-tRCD-tRP-tRAS):1-48-1-0 (cycles), 22ns read latency, 68 ns write latency
Power/Energy consumption	
DRAM	Voltage: 1.5V, Standby: 90 mA; Read and write on row bu,er hit: 160 mA and 165mA; Read and write on row bu,er miss: 200 mA
PCM	Read/write on row bu,er hit: 1.616 pJ/bit; Read and write on row bu,er miss: 81.2 pJ/bit and 1684.8 pJ/bit

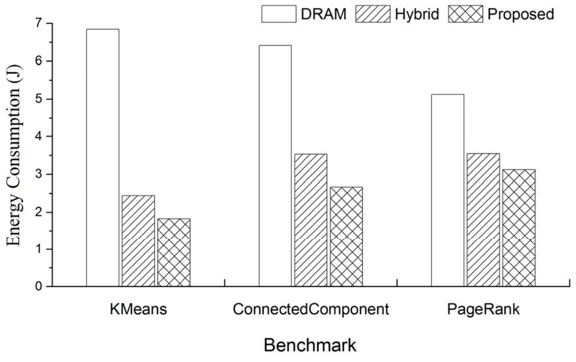


Fig. 7. Energy consumption.

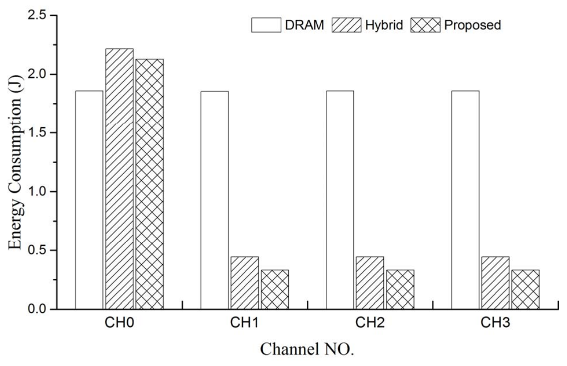


Fig. 8. Energy consumption distribution on each channel of one node.

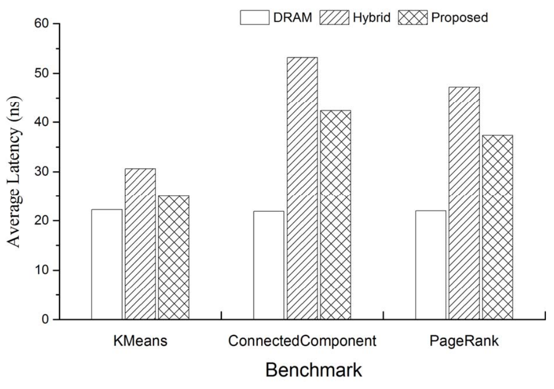


Fig. 9. Average latency of write and read.

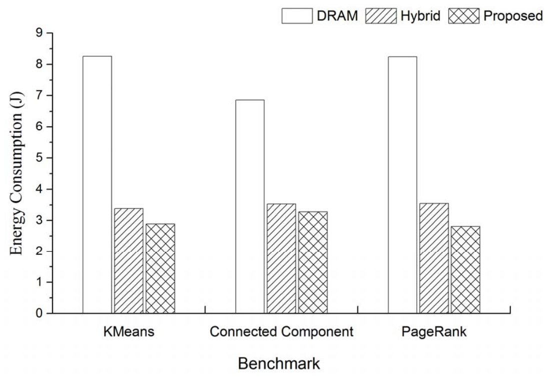


Fig. 10. Energy consumption in 2G memory.

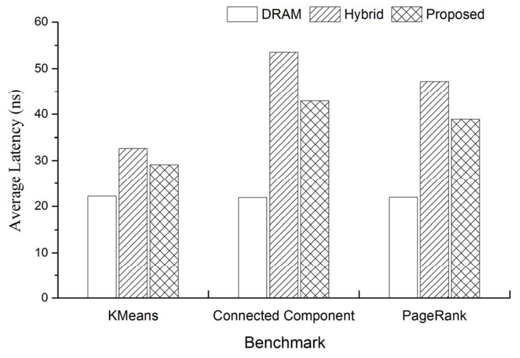


Fig. 11. Average latency in 2G memory.

According to our migration strategy, for cache blocks in DRAM, blocks with large active stage distance are migrated into NVM to reduce energy consumption. Conversely, blocks with small active stage distance remain in the DRAM because they will be accessed in the near future. As shown in algorithm 2, according to the migration strategy, once the active stage distance of a block in DRAM is bigger than the migration threshold, BlockManger will migrate it into NVM memory. As placement policy, the migration threshold can also be adjusted dynamically. The migration threshold adjustment is related to the available memory of the DRAM. For cache blocks in NVM, we migrate them in the similar strategy. Cache blocks in NVM with small active stage distance are migrated into DRAM. Due to space limitations, we have not listed the algorithm further.

D. Overall architecture

The proposed framework enables high performance, energy-efficient caching blocks in hybrid DRAM/NVM memory. And we will elaborate our framework details in following.

Architecture overview. As shown in Fig. 6, our implementation consists of six key components: (1) RCAnalyzer, lies on the driver node, analyzes the data dependency DAGs from DAGScheduler for recurring applications; (2) CacheManagerMaster, also lies on the driver node, is responsible for bookkeeping reference counts of all cache blocks; (3) Distributor, distributed on each executor node, implements the main logic of our placement strategy and determines which blocks would be stored into DRAM and which blocks would be stored into NVM; (4) Migrator, also distributed on each executor, implements the main logic of our migration strategy and determines which NVM blocks would be migrated into DRAM and which DRAM blocks would be migrated into NVM according their active stage distance; (5) Address multiplexer, distributed on each executor node, It is responsible for converting the address of the block according to the memory type of the block.; (6) Monitors, distributed on each executor node, collects cache statistics on worker nodes and report to the controller periodically.

Workflow. When BlockManager writes a cache block into storage memory, distributor will determine where it will be stored according to the placement policy. Then, the corresponding memory controller allocates memory according to the decision. Finally, BlockManager writes the block to the target memory. During execution, the migrator may migrate hot data block in NVM to DRAM based on the migration policy. In addition, some studies have implemented flexible ways to support NVM access. For example, [13] use off Heap mechanism to access NVM.

IV. Evaluation

In our experimental setup, we modified Spark 2.3.0 to collect memory access traces of block cache. The benchmarks run by Spark in a four heterogeneous nodes cluster. And the datasets are generated by SparkBench. Furthermore, HDFS2.6.5 is used for data storage with two replications a partition. NVMain is used to model components of DRAM and PCM, and memory hierarchy in detail. Our experiments setup on off-chip memories including PCM and DRAM, each has an individual memory controller and adopts channel-interleaving address decoding to exploit memory-level parallelism. And NVMain is used to evaluate power energy and latency of the benchmarks. Detailed configuration of our experiments is depicted in Table II.

We compare our approach, denoted as Proposed, with following memory systems. (1) Single DRAM, denoted as DRAM, uses DRAM in all channels. (2) Hybrid memory, denoted as Hybrid, uses a hybrid memory schema with one DRAM channel and three PCM channels. The proposed uses the same memory scheme as Hybrid.

A. Energy consumption performance

As shown in Fig. 7, our proposed framework can effectively reduce the energy consumption for data caching in Spark. Our approach consistently outperforms single DRAM memory across all three workloads. Compared to Hybrid, our approach achieves an approximate energy consumption in all benchmarks, especially having lower energy consumption in KMeans. Proposed spends more energy consumption on DRAM channel than Hybrid, as shown in Fig. 8. Compared to Hybrid, the proposed achieves 24.9%,24.5%,11.9% on energy consumption in Kmeans, ConnectedComponent and PageRank three benchmarks. And compared to single DRAM, it achieves 73.2%,58.5%,39.1% on energy consumption in the same benchmarks respectively.

B. Latency performance

Fig. 9 shows the access latency of three benchmarks. As expect, DRAM approach outperforms the other approaches in three benchmarks for low write and read latency performance of DRAM. But proposed approach has approximately access latency with DRAM in KMeans. Over three benchmarks, our approach prevails over Hybrid. Compared to Hybrid, Proposed achieves 18.4%,20.5%,20.9% on average latency in three benchmarks, respectively. The improvement in latency performance is attributed to write operations reducing of our approach within PCM channels.

C. Performance in memory-constrained situations

To further examine the scalability of the proposed framework, we experimented with the performance of the three schemas, under the circumstance of Spark with a memory-constrained situation. In this experiment, each channel of NVMain used a memory size of 512MB. As shown in Figures 10 and 11 the proposed framework is still superior to the other two solutions in a memory-constrained environment.

V. Related Work

In hybrid DRAM/NVM architecture memory, for memory heterogeneity, the data needs to be distinguished to place it into different memory. And a reasonable data migration policy can further improve system performance, such as access latency, energy efficiency, and NVM endurance. They all need data partition. Existing data partitioning strategies are categorized as hot/cold data partition and read/write data tendency partition [19].

Some strategies [8,9] with hot/cold data partition usually employ data access frequency or recent access interval to distinguish hot data and cold data. Typically, these strategies store hot data in DRAM to improve memory access performance and reduce NVM endurance, and store cold data in NVM in order to reduce energy consumption. For example, RaPP [8] proposed a rank-based placement strategies, simultaneously considers access frequency and recency. Pages in high rank are placed to DRAM and pages in low rank are stored to NVM. During execution, NVM pages with high rank queue are migrated to DRAM, and DRAM pages with low rank are schedule for migration to NVM. Trigger threshold considers page migration cost. Liu et al. [9] uses the number of accesses of pages in a slot as page hotness. Similarly, high hotness pages are placed to DRAM and NVM stores pages with low hotness. Its migration strategy adopted a utility-based approach, considering the read and write migrating gains and the cost of migrating. Moreover, it also uses a dynamic cache filtering strategy to determine the NVM pages that need to be migrated to DRAM. The pages with utility values are greater than the threshold would be migrated as DRAM, but instead is retained in the NVM. During the execution, filtering threshold dynamically adjustment to guarantee the effectiveness of the algorithm based on the change of page hotness in DRAM.

Other studies [20-23] with read/write data intensive partition classify pages into read intensive pages and write intensive pages according to the characteristics of page read and write times. The read intensive pages, which reading access is dominant, are stored in NVM, and write intensive pages are stored in DRAM. Lee et al. [20] compares temporal locality and frequency of write references, then concludes that write frequency is generally a better estimator than temporal locality in predicting the re-reference likelihood. Based on the observation, they allocate memory for missed data according to recent past write history. Read request data is stored to NVM, and write request data is stored to DRAM. LRU-WPAM [21] divides data by a dynamic preference weight based on each access type. And data migrates between DRAM and NVM based on the weight and a migration threshold. Additionally, MHR-LRU [22] predicts data intensive by tracking the number of write-hit in the DRAM and APP-LRU [23] uses a metadata information table that is historically accessed by a read and write access ratio to predict the tendency of data.

Moreover, some studies [13,24] have focused on supporting the memory computing framework for hybrid memory architectures. For example, Wang et al. [13] extends Spark to support for hybrid DRAM/NVM memory architecture by indicates NVM to Heap-off. The storage level of cache data that needs to be stored in the NVM would be set to a new flag, NVM_OFF_HEAP. Pan et al. [24] discusses a way to improve the Shuffle performance of Spark through NVM. Shuffle data is no longer stored in DFS as a file, but is persisted into NVM.

VI. CONCLUSION

In this paper, we present an energy-efficient data caching framework. Our framework uses a hybrid DRAM/NVM memory architecture that balances performance and energy consumption. Different from other low-level hybrid memory optimization methods, our method fully takes the characteristics of big data applications into account, and select the storage memory for the cache data at the application execution. Our approach makes decisions based on cache blocks. And fully consider the dependencies between data. Blocks with good temporal locality are preferentially placed in DRAM for execution, and data that is not accessed in time is stored in the NVM. Compared with the hybrid architecture memory with uniform data access and a single DRAM architecture memory, our approach can effectively balance performance and energy consumption. Experiments show that the proposed framework can effectively reduce energy consumption about 73.2% and improve latency performance by up to 20.9%.

ACKNOWLEDGMENT

Jie Tang is the corresponding author of this paper. This work is supported by South China University of Technology Start-up Grant No. D61600470, Guangzhou Technology Grant No. 201707010148, Guangdong R&D Key Project of China under Grant No. 2018B010107003, Doctoral Startup Program of Guangdong Natural Science Foundation under Grant No. 2018A030310408, and National Science Foundation of China under Grant No. 61370062, 61866038.

REFERENCES

[1] J. Hamilton, "Cooperative expendable micro-slice servers (cems):low cost, low power servers for internet-scale services," in Proc. the Conf. on Innovative Data Systems Research, 2009.

[2] Y. Yu, W. Wang, J. Zhang and K. B. Letaief, "LRC: Dependency-aware cache management for data analytics clusters, " INFOCOM 2017 - IEEE Conference on Computer Communications, 2017, pp. 1-9.

[3] C. Lefurgy, K. Rajamani, F. Rawson, et al. "Energy management for commercial servers." Computer 2003; 36: 39-48. DOI: 10.1109/mc.2003.1250880.

[4] Krishna T Malladi, Ian Shaeffer, Liji Gopalakrishnan, David Lo, Benjamin C Lee, and Mark Horowitz, "Rethinking DRAM power modes for energy proportionality, " In MICRO, 2012.G.

[5] Y. Xie, "Modeling, architecture, and applications for emerging memory technologies, " IEEE Des. Test, vol. 28, no. 1, 2011, pp. 44-51.

[6] G. Dhiman, Raid Ayoub, and Tajana Rosing, "PDRAM: A hybrid pram and dram main memory system," in Proc. the 46th Annual Design Automation Conference, San Francisco, California, pp. 664-469, 2009.

[7] M.K. Qureshi, V. Srinivasan, and Jude A. Rivers, "Scalable high performance main memory system using phase-change memory technology, " SIGARCH Comput. Archit. News, vol. 37, no. 3, 2009, pp. 24-33.

[8] L.E. Ramos, Eugene Gorbatov, and Ricardo Bianchini, "Page placement in hybrid memory systems," in Proc. the International Conference on Supercomputing, Tucson, Arizona, USA, pp. 85-95, 2011.

[9] H. Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo, "Hardware/software cooperative caching for hybrid dram/nvm memory architectures," in Proc. the International Conference on Supercomputing, Chicago, Illinois, pp. 1-10, 2017.

[10] J. Zhang, et al., "An optimal page-level power management strategy in pcm–dram hybrid memory," International Journal of Parallel Programming, vol. 45, no. 1, 2017, pp. 4-16.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. J. Franklin, S. Shenker and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, " in Proc. the 9th USENIX conference on Networked Systems Design and Implementation, 2012, pp. 2-2.

[12] K. Nguyen, L. Fang, G. Xu, et al. "Yak: a high-performance big-data-friendly garbage collector."in Proc. the 12th USENIX conference on Operating Systems Design and Implementation. Savannah, GA, USA: USENIX Association, 2016, pp. 349-365.

[13] Chenxi Wang, Fang Lv, Huimin Cui, Ting Cao, Zigman John, Liangji Zhuang, and Xiaobing Feng, "Heterogeneous memory programming framework based on Spark for big data processing, " Journal of Computer Research and Development, vol. 55, no. 2, 2018, pp. 246-264.

[14] Bo Wang, Jie Tang, Rui Zhang, Wei Ding, and Deyu Qi, "LCRC: A dependency-aware cache management policy for Spark" in Proc. the 16th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA-2018), 2018.

[15] M. Poremba, and Y. Xie, "Nvmain: An architectural-level main memory simulator for emerging non-volatile memories," in Proc. the 2012 IEEE Computer Society Annual Symposium on VLSI, 2012, pp. 392-397.

[16] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. "SparkBench: a comprehensive benchmarking suite for in memory data analytic platform Spark, " in Proc. the 12th ACM International Conference on Computing Frontiers (CF '15). ACM, New York, NY, USA, 2015.

[17] JinBao Zhang, "An energy management for hybrid memory based on write frequency of pages." MD di ss. Huazhong University of Science and Technology, 2015.

[18] Bo Wang, Jie Tang, Rui Zhang, Wei Ding, and Deyu Qi, "A dependency-aware storage schema selection mechanism for in-memory big data computing frameworks" International Journal of Parallel Programming, 2019.

[19] Zhangling Wu, Peiquan Jin, Lihua Yue, and Xiaofeng Meng, "A survey on PCM-based big data storage and management," Journal of Computer Research and Development, vol. 52, no. 2, 2015, pp.343-361.

[20] S. Lee, H. Bahn, S. H. Noh, "Characterizing memory write references for efficient management of hybrid pcm and dram memory," in Proc. the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, 2011, pp. 168-175.

[21] H. Seok, Y. Park, Ki Y. Park, Kyu Ho Park, "Efficient page caching algorithm with prediction and migration for a hybrid main memory," SIGAPP Appl. Comput. Rev., vol. 11, no. 4, 2011, pp. 38-48.

[22] Kaimeng. Chen, Peiquan. Jin, Lihua Yue, "A novel page replacement algorithm for the hybrid memory architecture involving pcm and dram," in, Berlin, Heidelberg, 2014, pp. 108-119.

[23] Z. Wu, et al., "App-lru: A new page replacement method for pcm/dram-based hybrid memory systems," in, Berlin, Heidelberg2014, pp. 84-95.

[24] Fengfeng Pan, Jin Xiong, "NV-Shuffle: Shuffle based on non-volatile memory, " Journal of Computer Research and Development, vol. 55, no. 2, 2018, pp. 229-245.