OPTIMIZING PERFORMANCE OF IN-MEMORY COMPUTING WITH HYBRID MEMORY SYSTEM

by

Wei Rang

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2021

Approved by:

_____
Dr. Dazhao Cheng

_____
Dr. Xi Niu

_____
Dr. Dong Dai

_____
Dr. Tao Han

ABSTRACT

WEI RANG. Optimizing Performance of In-memory Computing with Hybrid Memory System. (Under the direction of DR. DAZHAO CHENG)

The development of in-memory technologies has fueled the emerging of in-memory computing systems. Simultaneously, with novel memory technologies such as high-bandwidth memory (HBM) and non-volatile memory (NVM), hybrid memory systems are expected to be more commonly used in Cloud Computing platforms, which opens a new field about memory management in both academia and industry communities. Memory capacity is always a critical bottleneck for any applications running in Cloud Computing platforms. Data explosion is also posing an unprecedented requirement for computing capacity to handle the ever-growing data volume, velocity, variety, and veracity. Thus, in-memory computing systems are increasingly looking inward at huge caches of under-processed or trash-away data as resources to be mined. The key purpose of managing data in any memory system is to keep more useful data with high memory utilization without compromising applications' performance, especially for machine and deep learning applications running in clouds. However, there are numerous challenges in realizing the above goal, including sharing memory among applications, managing cached data, data migration on hybrid memory systems, and control strategy for a unified hybrid memory pool. In this context, we concentrate on developing an efficient hybrid memory system and memory management strategies for in-memory computing on Cloud Computing platforms.

To achieve this, we propose to develop a hybrid memory system that includes fast and relatively slow memory hardware and memory management strategies for applications running in cloud environments based on optimization formulations, feedback control, and machine/deep learning methods. In order to realize a runtime system that automatically optimizes data management on hybrid memory, we will (1) propose a new shared in-memory cache layer among parallel executors that are co-hosted on the same computing node, which

aims to improve the overall hit rate of data blocks by caching and evicting these blocks uniformly across multiple executors; (2) develop a middleware layer built on top of existing deep learning frameworks that streamlines the support and implementation of online learning applications; (3) design a unified in-memory computing architecture with efficient data sharing and communication strategy to optimize data migration and placement, memory allocation and recycle for machine learning applications. The problem of management of share cache memory including memory allocation and recycle will be defined as online optimization problems and solved with feedback control and machine learning algorithms in terms of memory utilization. To improve the utilization rate of memory for in-memory computing, we will (1) design an algorithm to predict the possibility of a cache data block to be referenced again and follows it to prevent blocks with longer re-reference distance from occupying the limited cache space too long time; (2) design a novel model updating strategy that builds training data samples in terms of contributions from different data life stages in model training, and considers the training cost consumed in model updating so that a better training model of describing data tendency in dynamic environments can be achieved; (3) design a memory management strategy for the hybrid memory system to automatically optimize data migration among different memory layers to achieve similar performance compares to the case of pure fast memory systems with a relatively limited capacity of memory size. We will implement these methods on top of some existing Cloud Computing platforms that aims to maximize memory utilization, holding more applications and less requirement for fast memory hardware. We will also implement experiments with the proposed technologies on a testbed of the local cluster environment and evaluate their performance with typical benchmark applications.

DEDICATION

This thesis is dedicated to my parents and wife for their constant support, encouragement and understanding in my life.

ACKNOWLEDGEMENTS

First and foremost I am extremely grateful to my advisor Dr. Dazhao Cheng for his continuous support of my Ph.D study and research, for his patience, invaluable advice, and immense knowledge. His guidance helps me in all the time of academic research and daily life. Besides my advisor, I would also like to thank the rest of my committee members: Dr. Xi Niu, Dr. Dong Dai, and Dr. Tao Han, for their insightful comments and encouragement, which inspires me to widen my research from various perspectives. I would like to thank my friend and lab mate Donglin Yang for a cherished time spent together in the lab, and in social settings. Finally, I would like to express my gratitude to my parents and my wife. Without their tremendous understanding, supports and encouragement in the past few years, it would be impossible for me to complete my study.

This work research and dissertation were supported by the NSF grants CCF-1908843 and CNS-2008265.

TABLE OF CONTENTS

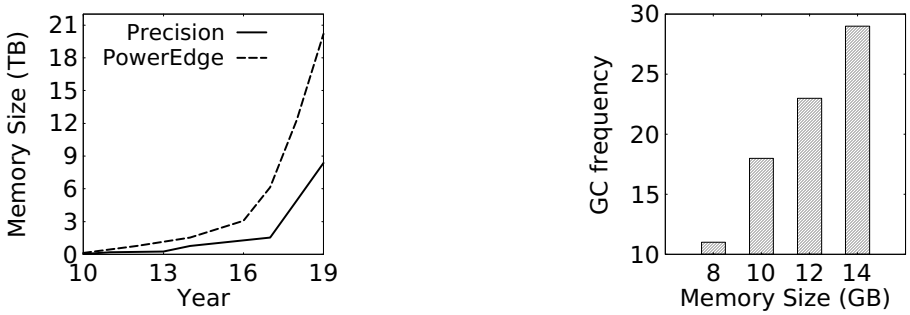
LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Memory plays a pivotal role in many popular distributed in-memory computing frameworks, such as Spark [1], Storm [2]. In such systems, frequent I/O operations can be significantly reduced so that application performance would be sped up by orders of magnitude via caching input and intermediate data into a specific memory space. To ensure proper memory utilization, a well-designed management strategy is essential and important for these in-memory computing frameworks, especially with increasing memory resources in cluster nodes (e.g., Precision and PowerEdge series of Dell servers [3] in Figure 1.1(a)).

Accordingly, many memory allocation strategies [4] have been adopted in data-parallel frameworks. For example, a huge amount of memory is preferred for a single executor on Apache Spark cluster in order to cache more intermediate data. In-memory computation is a significant feature of Spark platform so that computation efficiency could be further improved by avoiding frequent I/O operations between memory space and local disk. However, a large memory space assignment always increases Garbage Collections (GC) overhead for the JVM based in-memory computing framework, i.e., Spark. Our preliminary study in Figure 1.1(b) shows the number of GC times raise nearly 3X when memory



(a) Memory Capacity (b) GC frequency

Figure 1.1: Memory size and GC frequency.

size is increased from 8GB to 14GB. Given the fact that the execution process has to be paused when it is suffering from GC operations, setting a large memory space for individual executors is not always beneficial and should be avoided. It indicates a larger memory may not always guarantee a better performance due to frequent GC operations.

Another alternative solution is increasing the parallelism of task execution in such systems. It may deploy multiple executors on the same machine so that each of them will be allocated with a relatively smaller memory space. Although smaller memory may not cache as much data as larger ones do, this method can effectively decrease GC times, which has been widely applied in many systems [5] [6]. Such deployment of multiple executors also makes parallel granularity higher than the model of a single executor, which indeed accelerates the processing speed. However, the challenge is to design a fine-grained allocation policy as the input data sets of each executor is not identical. Furthermore, the memory demand in different computing stages of a process varies a lot over time. These two factors result in memory utilization imbalance among executors (i.e., tasks). Thus, we aim to fill in this gap by caching and managing intermediate data across multiple executors to improve and balance memory utilization.

With the popularity of in-memory computing and big data, Deep Learning (DL) has played an essential role in many practical domains, such as pattern recognition, recommendation system and data mining. Most DL applications are running in the environments where input datasets are dynamic streaming and data changing patterns are unexpected. For example, both speed and even syntax of people's speaking tone keep changing in conversations, user interests always shift in watching movies, climate data are frequently changing in weather forecasting. When facing these phenomena, also known as concept drift [7], predicting models based on static data becomes inaccurate and obsolete very soon, causing failures in future predictions.

Recently, there are a couple of approaches to tackle concept drift problems, among which a typical one is online learning [8]. It realizes continuously model updating with dynamic

data streams. Over the past decade, various online learning algorithms have been designed, e.g., Linear discriminant analysis [9] for pattern recognition, matrix factorization for recommendation system [10] and Bayesian inference for streaming data analytics [11]. With these algorithms, many issues on concept drift are solved and a range of applications are also developed in the industrial area. For example, Google [12] and Facebook [13] adopt online learning in predicting advertisement clicks. Netflix [14] uses a similar method in movie recommendation to its subscribers as well. However, the successful adoption of online learning is far less elegant since most online DL applications aim to improve the prediction accuracy with the sacrifice of code simplicity and easy-to-use interfaces. Furthermore, more hardware is expected to run such bloated systems, which significantly increase the system budget. Many popular DL frameworks, such as Coooolll [15] TensorflowOnSpark [16] never explicitly support for running online applications, and even less in model updating strategies. Currently, to achieve online learning purposes, users have to develop specific training loops to manually update models via a few basic strategies, such as continual and periodic updating approaches.

In order to increase memory capacity, hold more features for training DL models and decrease data migration between system memory and computing units, Hybrid Memory System (HMS) provides a promising solution to these issues. Within the HMS, different memory hardware components manufactured with various specifications are included to build the main memory and across all the computing units. Simultaneously, with novel memory technologies such as non-volatile memory, low-latency memory and high-bandwidth memory, hybrid memory systems are very promising in both academia and industry. A typical HMS usually includes multiple kinds of memory hardware with various features such as capacity, bandwidth and latency, which in turn raises the problem of data migration and storage. DL applications are always programmed with multiple execution phases and each running with unique working sets, which requires deliberate data management between various memory components to achieve better performance. The ideal case should be that

placing hot/frequent-accessed data into the fasted memory with lower latency and higher bandwidth, while other data is stored into other memory components.

Existing systems combine different memories in terms of capacity, latency, cost, bandwidth, and persistence to build a comprehensive hybrid memory system. The emerging of non-volatile memory (NVM) technologies attracts researchers propose more topics on hybrid memory systems. A representative usage of the NVM is to configure it as the extension part of the traditional DRAM [17] [18]. Many data placement and migration strategies have been proposed [19], [20], [21], [22], [23] to improve the overall performance of hybrid memory systems aiming to eliminate the huge performance differences between DRAM and NVM. To optimize a hybrid memory system for a better performance, one of the widely adopted solution is to place hot data in the DRAM until that space is full, the next operation is migrating unnecessary data into NVM so that the saved space can be used to host new hot data. As an application runs, its data would be assessed and migrated to achieve a better performance in terms of execution time. For the software-controlled hybrid memory systems, data access collection and migration requires effective software mechanisms and efficient strategies to determine data placement and migration [24] [25] [26].

Use Hybrid Memory System to reduce the size of fast memory component has been well studied in the a couple of works. Facebook adopts SSDs as a cache disk to reduce the memory footprint of databases [27]. Similarly, Bandana proposes a persistence memory system with SSDs to hold DL training models with system memory serving as a small cache [28]. SuperNeurons [29], moDNN [30] and vDNN [31] develop use heuristics methods to tackle data management in hybrid memory system between the CPU and GPU, which aim to overcome the limited memory capacity of GPUs. Some existing data management methods [32] [22] [33] adopt a sample-based approach to collect memory access patterns to decrease profiling cost. However, flash based SSDs systems [27] [28] use a software managed memory cache to overcome the poor performance caused by block-level management strat-

egy on NVM SSDs, while the data migration strategies [29] [30] [31] between GPU and CPU are problem specific heuristics that are less flexible and universal. To optimize data migration among different memory layers to obtain similar performance to the pure fast memory systems with a relatively smaller capacity in memory size, a lightweight system that automatically profiles and manages data with a fine granularity is necessary. Furthermore, this system should bridge the performance gaps among hybrid memory components, avoids unnecessary data migration and prefetches data into the right memory component when it is required.

A modern cluster running with hybrid memory systems are built from several nodes, each containing one or more computing units e.g., CPU, GPU, FPGA, a local DRAM and NVM, and remote DRAM and NVM [34]. To freely access all the memories across the whole cluster, Non-Uniform Memory Access time (NUMA) is a widely used solution. Accessing data hosted in local DRAM is much faster than that on the remote DRAM controlled by other nodes. This issue becomes more complicated in the hybrid memory systems due to these inherent performance differences in DRAM and NVM. Since NVM has a higher latency than DRAM, [35], hybrid memory system suffers more performance loss with the NUMA-based solutions which just evenly place data on DRAM and NVM. The traditional NUMA adopts the memory management strategy that evenly places data on DRAM and NVM to decrease the communication cost among nodes and improve memory access locality. This method is not much effective in the hybrid memory system with the fact that the accessing latency of visiting local NVM is much more higher than accessing the remote DRAM. Besides, the huge gap of accessing latency among DRAM and NVM plays a significant role in the total application performance. The traditional NUMA memory management strategy may even decrease the performance of applications in hybrid memory systems [36].

DNN has been dramatically successful over the past decade across many academic and industrial domains, including recommendation systems [37], real-time strategic game control [38], computer vision [39], and image synthesis [40]. In order to speed up the training

process of DNN, in-memory computing is introduced. However, the demands for higher model qualities come with more training data and larger model sizes, which result in larger memory footprints. For example, the latest models for language translation possess hundreds of thousands of parameters [41], which demand hundreds of GB memory space to hold the whole training network. The recommendation system developed by Facebook [42] includes orders of magnitude more model parameters than the traditional neural networks, which means tremendous memory space is demanded to guarantee the system's normal running.

To host the large DNN models and ensure a smooth training process, HMS profiles the memory access pattern of DNN training, bridges the performance gaps caused by different memory components and eliminate the adverse impact from data migrations. However, memory management becomes more complicated with taking the memory size and data scalability into consideration. In HMS, the fast memory size tends to be much smaller than the slow ones due to its high price. The average retail price of the DDR4 SDRAM (widely used as fast memory) increases by 2.3 times from 2016 to 2020 [43], motivating researchers to find an efficient memory management strategy for HMS. Moreover, data migrations between the fast and slow memories can be detrimental to the application performance due to the current training process has to suspend until the requested data is moved into the fast memory. Ideally, the data that is frequently accessed by DNN should be placed in the fast memory to ensure wider bandwidth and lower latency. In contrast, other less-used data is stored in the slow memory, which guarantees a better training performance and decreases data migrations.

Moreover, deep learning and big data applications are two hottest trends in the rapidly growing digital world [44] and has been dramatically successful over the past decade across many academic and industrial domains. To accelerate the training and predicting speed of these applications, in-memory computing is introduced. However, the requests for more accurate model come with larger memory footprints which is over the size of most DRAM.

Thus hybrid memory systems become a workable solution to this issue. Running deep learning and big data applications in hybrid memory systems further amplifies the asymmetric memory access latencies between DRAM and NVM, which demands a smart memory management strategy. Considering that the computation graph and data access pattern of deep learning and big data applications can be profiled at runtime, the overlapping of computation and data migration is possible. According to the profiled information, more data is to timely and proactively place in DRAM while some unnecessary data is moved to NVM.

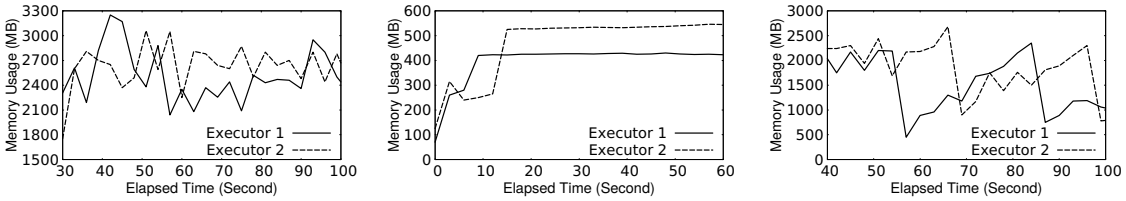
In this thesis, the main research contributions are summarized as follows: (1) design an algorithm to predict the possibility of a cache data block to be referenced again and follows it to prevent blocks with longer re-reference distance from occupying the limited cache space too long time; (2) design a novel model updating strategy that builds training data samples in terms of contributions from different data life stages in model training, and considers the training cost consumed in model updating so that a better training model of describing data tendency in dynamic environments can be achieved; (3) design a memory management strategy for the hybrid memory system to automatically optimize data migration among different memory layers to obtain similar performance compare to the pure fast memory systems with a relatively limited capacity of memory size. We implement these methods on top of some existing Cloud Computing platforms that aims to maximize memory utilization, holding more applications and less requirement for fast memory hardware. We also implement experiments with the proposed technologies on a testbed of the local cluster environment and evaluate their performance with typical benchmark applications.

CHAPTER 2: BACKGROUND AND MOTIVATION

We discuss the main motivations for Hybrid Memory System in cloud computing and our research focus in detail.

2.1 Applying Shared Memory among In-memory Computing Executors

Apache Spark is a popular in-memory computing platform with data encapsulated as an easy-to-use memory abstraction named Resilient Distributed Datasets (RDDs) [1]. A Spark cluster typically includes a master node and several slave nodes. $Executors$ are launched as JAVA processes on the worker nodes to execute assigned tasks. Each executor is allocated with specific on-heap memory space which is divided into many functional regions such as execution, storage and shuffle for RDDs operations. Meanwhile, a proportion of memory is allocated to executor as off-heap memory for caching blocks from on-heap memory. The state-of-the-art memory management strategy (i.e., $Unified\ Memory\ Management$) eliminates the existing boundary separating execution and storage memory regions in on-heap and off-heap memory space. Under this strategy, execution and storage memory could borrow some space from each other side as long as it is free. Unified memory management indeed helps increase memory utilization in both on and off heap space to a certain extent, but some issues still exist. If an executor is allocated with a huge amount of memory, especially for off-heap memory, too many GC activities would occur and decrease the performance of running tasks. The off-heap memory of individual executors is isolated so far in vanilla Spark and apparently inefficient given the fact that these memory demands of different executors vary a lot. Moreover, imbalanced memory utilizations and various requests of cached data in off-heap memory space make further harmful effects to system overall performance.



(a) Kmeans (b) SVD (c) PageRank

Figure 2.1: Imbalanced real-time memory utilization between two executors running the same workloads.

To quantify memory utilization with multiple executors under various heap sizes and their respective impacts on performance, an empirical study was conducted by using workloads from HiBench [45], a comprehensive benchmark suite. Spark-on-YARN mode is applied in order to flexibly configure executor numbers, CPU cores and memory sizes when running different workloads. We use Spark version 2.2 and Hadoop version 2.8.0 in the experiments. The default Least Recent Used (LRU) eviction policy is adapted for cache management. The cluster consists of 9 nodes, each of which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR4 RAM. All the nodes are interconnected via a 1000 Mb/s Ethernet. Each experiment was repeated for 5 times, and reported based on an average observation result.

2.1.1 Imbalanced Memory Utilization

Figure 2.1 depicts the memory usage traces of $Kmeans$, SVD and $PankRage$ under Case #2, especially imbalanced memory utilization reflected by gaps between trace lines. In this case, we launch two executors evenly to share the assigned hardware resource (each executor is allocated with 4 cores and 4GB memory). Figure 2.1(a) shows the memory usages of two executors for $Kmeans$ fluctuate over time and dynamically exist gaps between them. In the case of SVD, the memory demand increases very fast right after the workload starts running and then stays in a relatively stable memory usage status. The reason is SVD's CPU-bound property requires less amount of memory storing data but more CPU resource. Moreover, we find there are consistent gaps between the two executors after

Table 2.1: Resource Configurations

Case	#Executor	#Core	Memory	Total Resource
#1	1	8	8 GB	8 Cores & 8 GB
#2	2	4	4 GB	8 Cores & 8 GB
#3	4	2	2 GB	8 Cores & 8 GB
#4	8	1	1 GB	8 Cores & 8 GB

20_{th} seconds. The above observations demonstrate that (1) memory demands are dynamic over time; (2) memory usages between the two executors are imbalanced (shown as gaps between two lines). In particular, Figure2.1(c) shows $PankRage$ has a couple of peaks and valleys due to the suffering of GCs. These collapses between different executors result in low utilization and are detrimental to workload performance.

2.1.2 Impact of Parallelism

In our experiment, 8 CPU cores and 8GB memory were configured as the total available resource on each slave node to simulate a mainstream configuration.

As shown in Table 2.1, we set up 4 resource allocation cases. Taking Case #2 as an example, 2 executors were deployed on the same node with 4-Core and 4GB memory allocated to each executor. We ran three different workloads under each case individually, then the overall job runtime including garbage collection (GC) and computing time was recorded respectively.

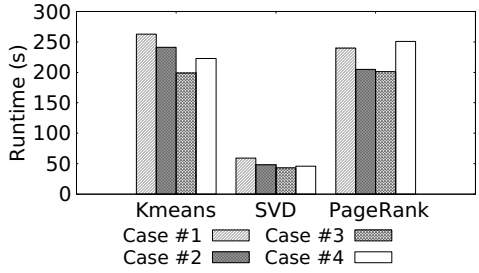


Figure 2.2: Impact of parallelism in terms of job runtime.

Figure 2.2 shows the runtimes of workloads ($Kmeans$, SVD and $PageRank$) under different resource allocation cases, i.e., with different execution parallelism. The results demonstrate the number of executors indeed has a significant influence on application per-

formance. On one hand, allocating all resource to only one executor cannot guarantee the best performance compared to the model of running multiple executors. On the other hand, too many executors also may hurt the overall job runtime. For example, the results of SVD and $PageRank$ under Case #4 cannot achieve significant performance improvement. Figure 2.2 shows the best performance is achieved in Case #3, #2 and #3 for $Kmeans$, SVD and $PageRank$, respectively. It demonstrates deploying two more executors brings a better performance gain than running a single executor in our case study. Thus, running multiple executors on a single slave node is necessary and beneficial, especially when the memory size of individual machines keeps growing recently.

2.2 Implementing Online Learning on Deep Learning Frameworks

2.2.1 Offline vs. Online Deep Learning

For many DL applications, the training stage is performed in an offline mode with batch data streams, which refers to an input dataset S = {s_1, s_2, s_3,...} with sample s_i = {x, y} consisting of an instance x and a target label y. The purpose of the training stage is to find a proper parameter θ for a model to predict a label y for each instance x correctly. Deep learning algorithms iteratively update θ over training samples, in an offline or online manner, depending on whether the whole training dataset is available or not. In online DL, there are new training samples available over time, so the model is updated according to such samples. Specifically, an online learning algorithm updates its parameter θ by θ = $f(\theta_i, s_i)$ when a new sample s_i is available, where f is an optimization function adopted in the algorithm. For offline learning, the parameter is updated with the entire batch S (containing a set of samples) of training dataset and the model is trained by iteratively applying an optimization function f' to all samples. That is, at the iteration of the training stage, the learning algorithm updates its parameter by $\theta = f'(\theta_k, S)$. Comparing these two learning strategies, it is evident that online learning is more lightweight and responses more promptly than offline learning as online mechanism only incorporates one data sample a time. This prominent feature makes online learning more efficient in dynamic environ-

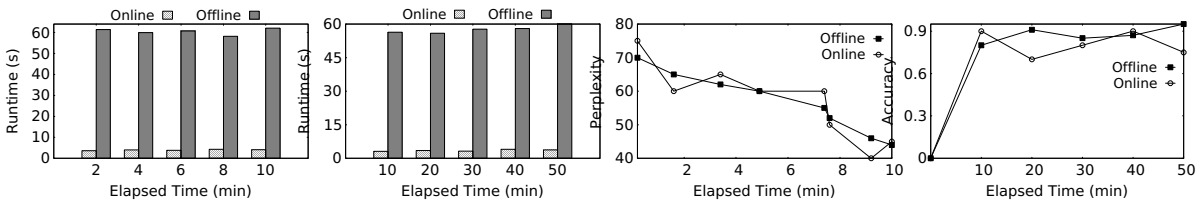
ments where the data source keeps changing over time, in which the trained model has to be frequently updated to accurately describe the trend of input data. In the following subsection, we demonstrate this benefit and possible weakness with case studies.

2.2.2 Case Study

We study two popular online DL applications, i.e., pattern recognition and classification prediction, which are based on Convolutional Neural Network (CNN) [46] and Deep Neural Network (DNN) [47] respectively. For pattern recognition, we adopt an MNIST [48] database to evaluate its performance on recognizing people's handwriting while using a 1GB dataset of user clicking activities to check efficiency in classification prediction.

To run these two applications, we adopt a testbed consists of 15 nodes, each of which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR3 RAM, running Ubuntu 16.04 LTS operating system with kernel version 4.0, Scala 2.10.0, and Hadoop YARN 2.8.0 for cluster management. One node serves as the *master*, and all the other 14 nodes serve as *slaves*. These nodes are connected with Gigabit Ethernet. We run these two applications on TensorflowOnSpark [16]. We use the built-in example of TensorflowOnSpark to evaluate performance over MNIST database. Another DNN study goes to classification prediction. This system decides what type of ads should be displayed to different users by predicting the probability of user click. We choose real-world traffic logs provided by Criteo [49] as the input data to evaluate performance. In the case study, we first train a base model using a small portion of the database and periodically update the model with dynamic input data. Then the performance differences in training time is compared between offline and online learning approaches. We finally evaluate how online learning improves model quality with continually generated data.

Training Time: Figure 2.3 (a) and (b) compare the training time achieved by online and offline training for CNN and DNN workloads respectively. In particular, we update the training model every 2 minutes in both online and offline manners. At each monitoring point, the offline method would retrain the model from scratch over all available data while



(a) CNN Training time (b) DNN Training time (c) CNN Model quality (d) DNN Model quality

Figure 2.3: The comparison of training time and model quality achieved by offline learning and online learning.

the online mechanism only takes in the new arrival data. These processing logics result in the orders of magnitude difference between the two methods. It is evident that the online training is significantly faster than the offline training. The reason behind is that offline training takes orders of magnitude longer time to retrain the model in an offline manner by incorporating all historical data. In contrast, online training only uses new arrival data and does not need to wait for the moment that all data are available.

Model Quality: Figure 2.3(c) and (d) display how the incorporation of new arrival data improves the model quality. In Figure 2.3(c), we adopt a popular metric named *perplexity* [50] (lower is better) to measure the model quality of CNN application. We measure the training model quality every 2 minutes in offline and online manners respectively. With more data fed into the model, both perplexities decrease, which means a better model quality. Figure 2.3(c) shows the overall perplexities of offline and online trainings are very close while online manner is not as stable as offline training. We then compare the model performance of online and offline training in overall prediction accuracy. This generally used metric quantifies the overall prediction performance of a classification algorithm (larger is better). Figure 2.3(d) shows that the model quality by offline training increases with more data get incorporated, and its quality trend to be stable. The model quality by online training outperforms offline training at the time points 10 and 40 while it is worse than offline training during other time periods. The reason is that only using new arrival data in the online learning approach cannot guarantee stable model quality compared to the offline learning approach. Although such naive data update strategy contributes to speed up the

data training process, it may discard many data samples with vital information to improve the model quality.

2.3 Scalable Data Management on Hybrid Memory System

In this section, we first provide a basic knowledge on the training/learning process of typical DNN models, and then a case study is offered to drive the motivation of this section.

2.3.1 Graph Based DNN Training

DNN's training stage often contains an optimizer and a backward propagation algorithm. A typical DNN model usually contains a couple of layers, which is comprised of a set of neurons. Each neuron of every layer conducts a non-linear function based on the neurons' outputs from the preceding layer, using a set of weights [51]. Training DNN models often include many iterative steps, which involve a set of training data objects input into DNN. Training DNN with deep learning frameworks e.g., TensorFlow [52] implements DNN as a computation graph with a set of nodes or vertex, which represent some computational kernel. Each kernel is defined with a couple of attributes e.g., the number of inputs and outputs, computation complexity, and computation time. Data dependencies and control between kernels are expressed as directed edges of computation graphs. Edges representing data dependencies are assigned with a tensor that takes contiguous fixed memory space size. Moreover, the computation graph is static and repeated in the whole training process. These features make profiling the memory access pattern of data objects possible and then optimize data migration in terms of the static DNN computation graph.

There are mainly three steps involving in data management on Hybrid Memory System (HMS): (1) memory access profiling, which is responsible for collecting memory access information for data objects or memory pages (2) decisions for data migration, which follows performance optimized models or cache management algorithms to determine the suitable data objects or pages to migrate to achieve better performance, and (3) data migration, which includes data placement and data movement with the purpose of decreasing data

migration cost.

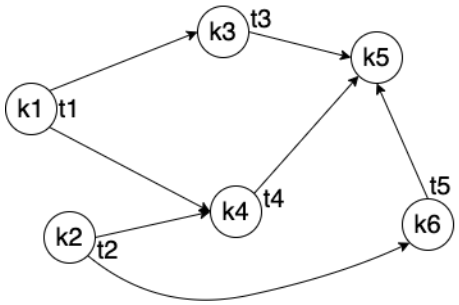


Figure 2.4: A sample of computation graph.

Figure 2.4 depicts a typical example of computation graph configured with 6 kernels and 5 tensors. Nodes in the computation graph represent computation kernels while edges means tensors consumed by kernels. Nodes are denoted with 0 or more inputs or outputs. The inputs or outputs of each kernel are also tensors. Every tensor is displayed by its producer kernel, all its consumer, and also its last consumer. When the last consumer finish its computation, the memory of this tensor can be freed to other coming tensors. For example, tensor *t2* is produced by kernel *k2* and it is consumed by kernels *k4*, *k6*. The memory space occupied by *t2* cannot be released until *k6* finished its computation, which means *k6* is actually the last consumer of *t2*.

In this thesis, we concentrate on the case that the computation graph describing a static DNN training iteration. That is, there is no data dependencies in the computation graph and its static graph structure is known at compile time. Moreover, the total sizes of intermediate data is known with the compiling process. In this case, we can predict and record the memory access behavior of each tensor and migrate it into the corresponding memory component to achieve better performance.

2.3.2 Case Study

We characterize and analyze the memory access pattern on DNN, and adopt the observations to motivate our work. We adopt Persistent Memory Block Driver (PMBD) [53] to simulate the slow memory. PMBD is a PCIe NVM device simulation based on DRAM.

200ns delay for the read/write operation and 19GB/s bandwidth are set up to simulate slow memory media. We use the DRAM as the fast memory, which is configured with 34GB/s bandwidth and 90ns latency.

We adopt the ResNet50 V2 workload to analyze data objects (tensor) and their access patterns. Only one training step is used for profiling this information. ResNet50 V2 is fed with the CIFAR-10 data set with 64 layers and 128 batch size within a forward and backward pass. Besides, in this case study, we only use the regular DRAM which means there is no fast or slow memory included. A data object is alive from the moment that it is allocated into the memory until its memory space is freed. Base on the concept of alive, we define the liveness of a data object as in how many layers the data object is alive.

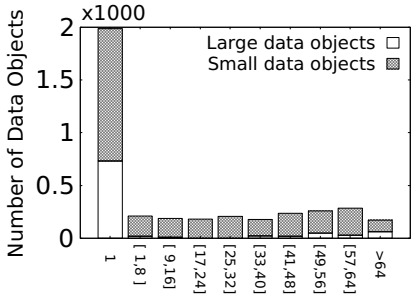


Figure 2.5: Distribution of liveness of data objects and data sizes from ResNet50 V2.

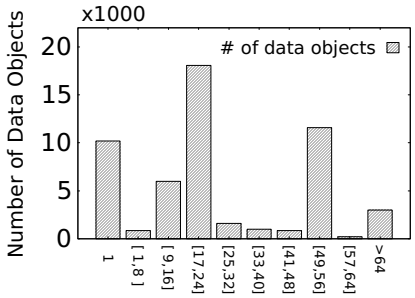
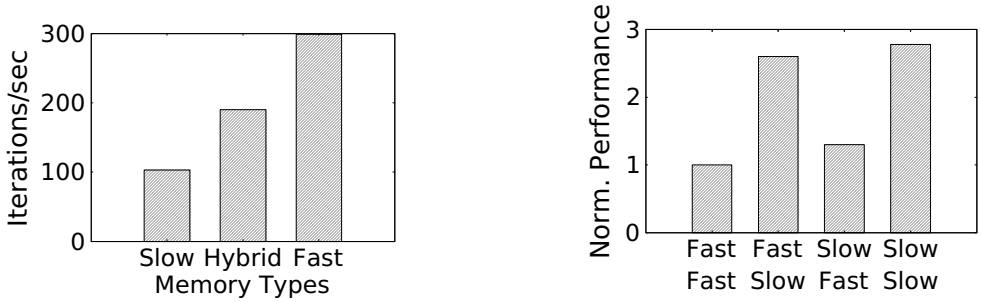


Figure 2.6: Distribution of memory access at the layer distance level.

Memory Access Pattern: Figure 2.5 displays the distribution pattern of data objects' liveness and the ratio of data sizes. In this case, we regard the data object with size smaller than 4KB as small data object while the remaining objects as the large data object. Y-axis



(a) Performance of iteration.

(b) I/O performance of memory.

Figure 2.7: (a) shows the iteration performance of ResNet50 V2 with a batch size of 128. (b) plots normalized performance to all I/O committed in fast memory.

denotes how many layers (liveness) a data object can survive in the model training step, the last label ">64" means that these data objects crosses more than one full training step. Figure 2.5 shows that more than 90% of data objects cannot live more than one layer, which means their liveness are shorter than a layer. Moreover, within those data objects, 97% of their size is smaller than 4KB (small data objects). In this paper, we define the data object whose liveness is shorter than a layer as short-lived data object and the data objects can survive more than one layer as long-lived data object. By hosting short-lived data in fast memory space, we can decrease unnecessary data migrations, increase memory utilization and improve DNN training performance.

Figure 2.6 depicts the distributions of memory access at the layer distance level. The figure also displays a huge amount of data objects are accessed in the first 24 layers. Among those data objects, nearly 80% of them are accessed in Layer 9-24. Those are the frequently used data objects that should be placed in the fast memory. In contrast, some data objects are less accessed. For example, in the layer range [57, 64], there is almost no data access occurred. The uneven distribution of data objects and their unbalanced access pattern in DNN provide opportunities for data management.

Impact of Different Memory: We conduct a further case study on the performance of ResNet50 V2 with 128 batch size under different memory settings. PMBD [53] simulates a slow memory while DRAM serves as the fast memory. Three memory settings are config-

ured: all slow memory, hybrid memory and all fast memory. We set up 10GB in slow-only and fast-only memory cases, 2GB fast memory and 8GB slow memory in the hybrid case.

Figure 2.7(a) shows the training performance for three different memory settings: the slow-only memory, the fast-only memory and the hybrid memory running with NUMA [54]. The Slow bar displays that simply replacing fast memory with slow memory cause poor application performance (about 3x slowdown) in training steps. The Hybrid bar shows that some specific optimized data management for hybrid memory is necessary, providing only a limited improvement over the Slow case. The Fast case brings the best performance with almost 3x and 1.5x speedup compared with Slow and Hybrid cases respectively, but the cost is more expensive than the first two settings. Thus, a hybrid memory system equipped with a smart data management strategy could reduce the performance gap between slow and fast memory.

The read/write speed of memory has affections on the performance of kernels with their inputs and outputs hosted in slow or fast memory hardware. We analyze the performance behavior with a single CONV kernel from ResNet50 V2. Figure 2.7(b) demonstrates the execution/computation time of this kernel with its inputs (upper label) and outputs (lower label) saved into Slow and Fast memory hardware. We notice that with the inputs to this CONV kernel are hosted in the Slow memory and its outputs are saved in the Fast memory, its performance is very similar to the case when both inputs and outputs are in Fast memory. But in the case when the outputs are saved in Slow memory, the kernel suffers a performance loss dramatically with over 2x slower. This behavior inspires us to consider the I/O latency differences when deciding where to place a data object to get an optimal runtime with a fast memory size constraint.

2.4 Memory Management Strategy on Unified Hybrid Memory System

Modern hybrid memory systems are usually configured with the traditional DRAM and non-volatile memory (NVM) [54] hardware. Non-Uniform Memory Access (NUMA) is the architecture adopted to manage data saved into these hybrid memory systems. Com-

pared with DRAM, NVM brings higher storage capacity and lower power cost at the expense of slower access speed and lower memory bandwidth. The commercial NVM, such as Intel Optane DC Persistence Memory [35], provides a max of 6.6 GB/s read bandwidth and 2.3 GB/s write bandwidth. Its maximum read latency is 346 ns, about 3 times higher than DRAM. Specifically, the bandwidth of randomly writing operation in NVM is around 30 times slower than that in DRAM. Based on those I/O differences, we use Persistent Memory Block Driver (PMBD) [53] to simulate the performance features of NVM via splitting part of the DRAM, which offers a flexible settings in memory hardware specifications.

2.4.1 Case Study

We characterize the issues of using NUMA memory management strategy in hybrid memory system running on our local cluster, which has four nodes equipped with Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR4 DRAM. According to the NVM features discussed above, the read and write latencies of NVM are configured to be 3X and 8X of the DRAM, and the bandwidth of NVM is limited to be one half of the DRAM. In this case, there are basically four kinds of memories in the hybrid memory system: local DRAM, remote DRAM, local NVM, and remote NVM.

We select Terasort [55] and ResNet152 [56] as the typical workloads from big data processing and deep learning applications, respectively. The Terasort illustrates uniform memory access pattern on all data, while ResNet runs on with different data access behaviors in terms of varied access frequency. We adopt Intel Memory Latency Checker (MCL) [57] to quantify the communication cost among nodes. We compare two NUMA memory management strategies: (1) The default NUMA interleaving policy, which evenly distribute data pages on DRAM and NVM. (2) Manually set the size ratio of NVM-to-DRAM to be 1:4, resulting in more data is to be hosted in DRAM. Both of the policies follows DRAM serves first policy, which means NUMA preferentially places data on DRAM when there is still some free space available, and then places data on NVM. Besides, the automatic NUMA

balancing (anb) policy is also included to further describe NUMA performance in hybrid memory systems.

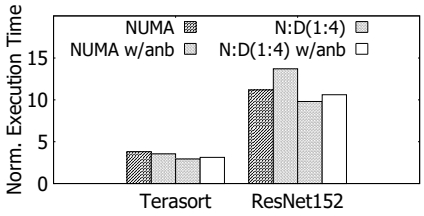


Figure 2.8: The execution time of two selected workloads with four different data placement strategies, all the results are normalized to the case that only allocates data in DRAM.

Workloads Execution Time: Figure 2.8 shows the execution time of Terasort and ResNet152 using NUMA (the default page interleaving data placement strategy), NUMA with the automatic NUMA balancing (anb) policy, the case of configuring the ratio of NVM-to-DRAM to be 1:4, and the case of NVM-to-DRAM (with the ratio of 1:4) with the automatic NUMA balancing (anb) policy. To provide a direct comparison, all the results are normalized to the case that only allocates data in DRAM. Figure 2.8 shows that the NUMA cases usually fails to achieve the best/optical application performance in the hybrid memory system, no matter it is configured with the automatic NUMA balancing (anb) policy or not. The policy NVM-to-DRAM ratio as 1:4 brings a better execution time compared with the NUMA strategy for both Terasort and ResNet152. However, the case of NVM-to-DRAM by a ratio of 1:4 with the anb policy performs a little bit worse than the pure NVM-to-DRAM case due to the bandwidth gap between DRAM and NVM, which takes more time in data migration. Overall, the default data placement strategy in NUMA that evenly places data on DRAM and NVM may hurt the performance of application in hybrid memory systems.

Data Locality: Figure 2.9 shows the tendencies of data localities in DRAM of Terasort and ResNet152. We select a 25-second time window when the workloads start to run. Data localities fluctuate a lot with application running on, Terasort displays a relatively stable tendency due to its uniform data access pattern while ResNet152 has more peaks and

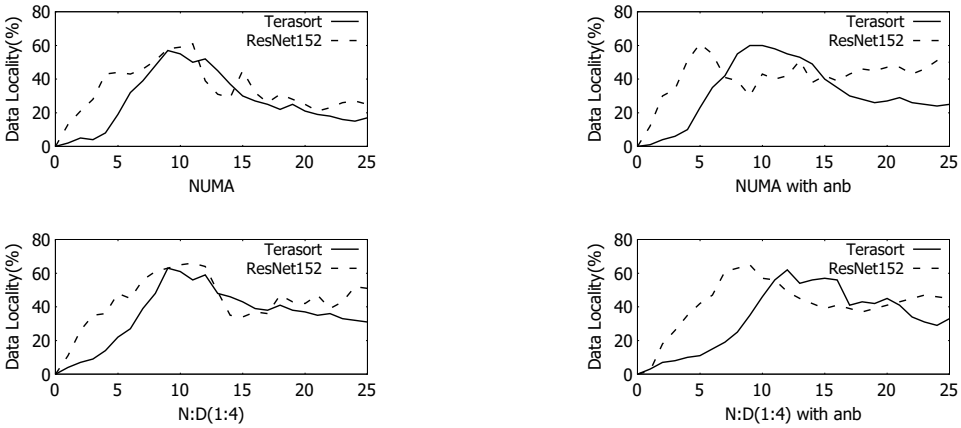


Figure 2.9: The tendencies of data localities in DRAM with different data placement strategies, all the data is from a 25 seconds time window when the workloads start to run.

valleys with different hot data paged in and out. But the average data locality of ResNet152 is higher than Terasort. The anb policy can improve data locality by dynamically migrating data to the node that the workload is running. We can also observe that the anb policy indeed help increase the data locality both in NUMA and NVM-to-DRAM cases. However, higher data locality may not guarantee better application performance. When compares Figure 2.8 and Figure 2.9, it is obvious that the cases with the anb policy do not bring the best execution time no matter the anb policy is configured with NUMA or NVM-to-DRAM. In the case NVM-to-DRAM with the anb policy, its execution time is much longer than the pure NVM-to-DRAM strategy. Because the anb policy wrongly migration data from DRAM to NVM, which has a slower bandwidth and higher access latency.

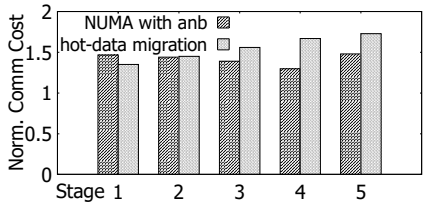


Figure 2.10: The total communication costs of intra-nodes and inter-nodes of ResNet152 from NUMA with anb and hot-data migration strategies, which include data migrations among local DRAM, remote DRAM, local NVM, and remote NVM.

Communication Cost: We analyze the total communication cost among different memory media from ResNet152. The total communication cost consists of inter-nodes cost and intra-nodes cost measured by Intel Memory Latency Checker (MCL) [57]. Figure 2.10 illustrates the total communication cost from NUMA with the anb policy and hot-data migration strategy in five stages divided by ResNet152's whole runtime. We choose the case of NVM-to-DRAM with a ratio of 1:4 as the baseline and all the results in Figure 2.10 is normalized to it. We notice that both NUMA with the anb policy and hot-data migration strategy migrate a huge amount of data form DRAM and NVM, and thus bring a lot of performance loss with failing to use the DRAM's higher bandwidth. The tradition anb policy migrates data among different memory media following the mostly recently access strategy. But the inherent performance gaps between DRAM and NVM make the anb policy less efficient and bring more communication cost. The hot-data migration policy uses a pre-defined threshold to migrate data, which could bring a better communication cost in some stages. But the hot-data migration policy does not consider the overall bandwidth utilization and the hybrid memory system equipped with DRAM and NVM makes it less useful.

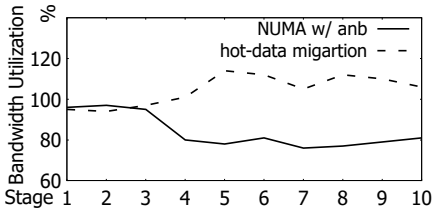


Figure 2.11: The bandwidth utilizations of ResNet152 using NUMA with anb and hot-page migration, all normalized to the case with the ratio NVM-to-DRAM set to be 1:4.

Bandwidth Utilization: Figure 2.11 shows the bandwidth utilization of ResNet152 in ten running stages divided by execution time using NUMA with the anb policy and hot-data migration policy, all the results are normalized to the case of NVM-to-DRAM. We find that both the anb and hot-page migration policies have a good bandwidth utilizations at the first three stages. But their performances vary a lot with workload running. The

anb policy shows a less efficient bandwidth utilization due to it merely tried to evenly place data on NVM and DRAN. Moreover, the anb policy brings more data migration from DRAM to NVM, which is actually moving data from memory with high-bandwidth to low-bandwidth, and thus fails to take advantages of the high bandwidth of DRAM. The hot-data migration strategy improves the utilization of memory bandwidth after Stage 3 and then its utilization drops in the Stages of 7 and 10. The reason is that the hot-data migration strategy fails to consider the memory bandwidth utilization when performs data migrations. The data distributions in hybrid memory system is more complicate with different memory bandwidth, we should design a smart data placement and migration policy to achieve higher memory access performance.

2.5 Challenges

We discuss several challenges encountered in optimizing performance of in-memory computing with Hybrid Memory System.

2.5.1 Memory Management on Shared Cache Memory

These observations from Section 2.1 show the impacts of running multiple Spark executors on a single machine: (1) The memory utilization discrepancies among executros inspire us to propose a more efficient cache management policy among multiple executors without losing performance as well as ensuring efficient memory utilization; (2) Deploying multiple executors indeed speed up the performance of application in some cases, but it cannot always bring performance imprivement e.g., *SVD*'s performance illustrated bu Figure 2.2. In the cases that multiple executors achieve positive effects, how to build the shared cache memory, how much memory each executor should share and how to reallocate the shared memory space among executors are the key points that need our attention.

Specifically, we need propose a new shared in-Memory cache layer among these parallel executors which are co-hosted on the same slave machine in Apache Spark. It aims to improve the overall hit rate of data blocks by caching and evicting these blocks uniformly

across multiple executors. The critical insight of this layer is to develop a novel eviction strategy to efficiently manage the shared cache space across executors to maximize overall cache hit rate as well as application performance. The new eviction policy can predict how far a data block will be referred again and consider all cached blocks to choose the ones that have little possibilities shortly. By leveraging global data referring information, the data management strategy is capable of evicts=ing less possibly used data and making more free space for requested data blocks.

2.5.2 Online Learning Implementation on Deep Learning Frameworks

The case study in Section 2.2 has demonstrated the benefits and shortcomings of online DL with dynamic batch data streams. Although online training defeats offline manner in terms of training time, the model quality brought with online training is not always as stable as offline approach. Thus, a trade-off is necessary to achieve the accurate model quality and the low training cost simultaneously. Indeed, the data contributes diversely to the model quality in model updating stage. Given the fact that the model updating must be performed in an online manner and without prior knowledge of future data, there is an imperative need to develop a strategy to combine and update data based on its real contribution to the model quality of online DL applications in dynamic environments.

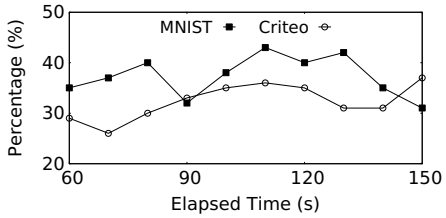


Figure 2.12: Duplicated data occurrence in a time window.

Figure 2.12 illustrates the percentage of data samples with duplicates during the model updating process. It is evident that the duplicated data take a relatively high percentage in the training dataset. Such behavior inspires us to take data occurrence into consideration when updating the training model. Ideally, we should decrease the latency of data

incorporation with delicate combined data samples and obtain a model with high quality at low training cost. In light of the above issues, our prime goal is to develop a system that friendly supports online learning applications without much coding modifications and hardware resources. To be exact, we plan to introduce an easy-to-use middleware solution based on existing DL frameworks to streamline the support and implementation of stream-based workloads. Moreover, an efficient model updating strategy is necessary to determine how and when to perform model updating for DL applications considering data life stages and training costs.

2.5.3 Memory Management on Hybrid Memory System

The case study in Section 2.3 has demonstrated the uneven distribution of short-lived and long-lived data objects and the unbalance access pattern across DNN training layers. Besides, the read/write speed asymmetry from the fast and slow memory results in large implications on kernel's performance, especially when a kernel places its outputs in the slow memory. Given that the DNN computation graph is static and its memory access pattern is stable, we can profile each tensor's data behavior to build a scalable memory management strategy for HMS, which is smart in controlling data placements and migrations. In the ideal case, short-lived data objects are to placed in the fast memory while long-lived ones in the slow memory. Specifically, the input data location should be considered because it has a huge effect on DNN training performance.

2.5.4 Memory Management on Unified Hybrid Memory System

From Section 2.4, we can observe that the traditional NUMA memory management strategy and its widely used anb policy are not efficient in hybrid memory systems. The case study has demonstrated the higher application execution time and communication cost, and lower memory bandwidth utilization. Besides, the fluctuant data localities make data migration more complicate. Given that a hybrid memory system is usually equipped with different memory media with various features in bandwidth, access latency, and capacity.

The above observations motivate us to propose a smart data management mechanism to achieve better performance for applications deployed in hybrid memory systems.

2.6 Objectives and Contributions

In this thesis, we will design and implement novel memory management strategies in shared cache memory and hybrid memory system to optimize in-memory computing performance. The objectives and expected contributions of our research are as follows:

1. *Shared Memory Cache Layer for Multiple Executors*: We design a shared memory cache space, i.e., iMlayer, which is deployed between on-heap memory and local disk, to cache and manage intermediate data across multiple executors so that I/O operations can be decreased. A fuzzy model is adopted to decide memory donation from each executor to iMCache, the shared cache memory, and implement a novel cache eviction policy named Next Re-reference Distance (NRD) which aims to improve the overall hit rate of data blocks by caching and evicting these blocks uniformly across multiple executors.

2. *Model Updating Strategy for Online Deep Learning*: We tackle these challenges in Section 2.2 with iDlaLayer, a thin middleware layer prototype atop DL frameworks (e.g., TensorflowOnSpark) that facilitates stream-based online learning applications. We introduce a concept named *data life cycle*, with which streaming data is divided into various life stages in terms of how much contributions it made to model updating. We also propose a novel data life aware updating strategy (DLA), which relies on combined data sample and considers training cost when deciding whether to perform a model updating action.

3. *Data Management on Hybrid Memory System for Deep Leaning*: We design and implement a middleware that automatically determines the optimal data migration strategy and exploits domain knowledge on DNN to decide data migrations between the fast and slow memories in hybrid memory system. To achieve a better performance in data migrations for DNN training, we introduce a reference distance and location based data management strategy (ReDL) that treats short-lived and long-lived data objects with Idle and Dynamic migration methods, respectively.

4. *Unified Hybrid Memory System for In-memory Computing*: We propose a unified memory system across the cluster named UniRedl, which automatically optimizes data migration in hybrid memory system based on data access pattern and computation graph of applications. UniRedl adopts the primary/replica mode on a cluster to abstract all the memory into a uniform space address. To achieve a better performance for big data and deep learning applications, we introduce a novel memory management strategy that considers data location, memory capacity&utilization and computation graph.

CHAPTER 3: RELATED WORK

3.1 Memory Management for In-memory Computing

Memory management is a well-studied research topic while application's performance has been widely improved by various memory strategies [58] [59]. These systems refer to either change workload scheduling or apply different heuristics on platform management. In [60], the memory model is assumed to be a small cache. Memory access activities of given applications are categories into various phases. From such memory accesses breakdown, the system-level evaluations are made among any two consecutive commands. Although this approach offers a good insight into the benefits of in memory computing in the system level, it fails to consider the situation of hierarchical memory and data locality. As an alternative, the work done by [61] takes multi-level caches into consideration. Furthermore, this work also studies data locality in order to determine whether it is worthy to transfer some computation to the memory unit. However, this work assumes that the users have enough knowledge on the application, and can manually decide which parts of application could deployed in memory computing.

Although Spark job performance can be improved by increasing execution parallelism, this strategy may introduce additional overheads by GC and I/O operations. Thus, many works focus on optimizing the deployment model of multiple executors/JVMs in Spark. Emerging JVM technologies such as heap ballooning [62] and dynamic heap sizing [63] provide mechanisms to release committed memory from the virtual heap space. Tachyon [64] provides an in-memory distributed caching layer to cache intermediate data across different frameworks. The key idea is using distributed shared memory layer to store data blocks. Similarly, Tungsten [65] proposes a method to change memory management of JVM from on-heap to off-heap space. However, it is difficult to decide how much memory

should be allocated to each executor and which data blocks are supposed to be evicted if any memory tension occurs. In contrast, iMlayer aims to build the same-functional shared memory layer for the local executors on the same slave machine, which offers a convenient way to manage cached data without a centralized server for saving and synchronizing data location information.

Moirai [66] focuses on cloud resource allocation with performance isolation in terms of requests per time window while Ginseng [67] is for memory pricing and auctioning cloud platforms. These works only focus on pricing memory resource for applications that have a specific shared cache memory server running on VMs. In contrast, iMlayer enables multiple executors to share a dedicated cache space and ensures a higher hit rate via a novel cache eviction policy, i.e., NRD. Moreover, our work focuses on managing the local memory resource on each slave machine, which is more scalable for most distributed systems.

3.2 Implementation Methods to Online Deep Learning

Various deep learning frameworks are available to provide proper support for popular neural network algorithms, such as TensorFlow [52], Coooolll [15]. These platforms offer fully functional libraries and APIs for DL algorithms. Recently, a new trend in deep learning is to realize distributed learning with a parameter server model. BigDL [68] allows end-users to build deep learning applications using a single unified data pipeline and the whole pipeline directly run on top of some existing deep learning systems in a distributed fashion. Geeps [69] is another framework that implements deep learning with large volumes of data on distributed GPUs, it incorporates and extends the parameter server training model. However, most of them perform well in offline training models while never explicitly supporting online updating strategies. Instead, users have to design customized training loops to manually realize online learning via continual or periodic approach, which is inefficient and cumbersome. iDlaLayer is orthogonal to these systems and complemental to them with supporting and implementing online DL applications.

Data streaming systems have been widely studied and increasingly used in real-world commercial solutions rather than pure academics. Differential Dataflow [70] is designed to process large volumes of streaming data efficiently and to respond to arbitrary changes in the input collection quickly. Naiad [71] automatically incrementalizes dataflow computations and is capable of building low-latency data-parallel iterative computation. However, Differential Dataflow [70] and Naiad [71] mainly focus on quickly react on larges input data and respond to any changes inside the data in the context of a declarative data-parallel dataflow language, and cannot respond to online learning cases that processing the same computation with different data sample in each iteration. Spark Streaming [72] allows users seamlessly intermix streaming, batch and interactive queries. SECRET [73] is a descriptive model that enables users to analyze the behavior of systems and understand the results of window-based queries for a broad range of heterogeneous SPEs [73]. MillWheel [74] provides a programming model with a notion of logical time, good scalability and fault tolerance, making it simple to write time-based aggregations. iDlaLayer is an online learning system for stream-based workloads in dynamic environments. Similar to studies [75] [73] on latency and throughput in processing streaming data, our algorithms can help such systems achieve a balance between processing cost and latency.

Recently, many works focus on exploring deep learning model updating strategies. For example, a recent report [7] provides a thorough survey on concept drift in the scenarios of online learning with streaming data. Incremental&Decremental SVM [76] and Weighted-SVM [77] resort to re-engineer existing learning algorithms by developing incremental versions of the basic SVM and adjusting the training data sample weights in an SVM-specific manner, respectively. InferLine [78] is designed to provision and manage deep learning inference pipelines for latency-sensitive applications with cost-efficient feature. It consists of two principal components that operate at time scales orders of magnitude apart to configure the system for near-optimal performance. Velox [79] proposes a model updating strategy with combing online learning and statistical techniques. DLA is complementary to these

works, and could potentially be applied in a system like Velox to support online learning, improve training model quality by continually incorporating streaming data and strike a balance between model quality and training cost.

3.3 Data Management Policies on Hybrid Memory System

Many works [80] [81] [82] have been proposed to build heterogeneous main memory system. Specifically, with the emerging technology Intel Optane DC, several recent works [35] [83] [84] explore the combination of Intel Optane DC and traditional DDR to build a hybrid memory system. [35] explores the specifications, features, and performance of Intel Optane DC persistence memory. It also studies Intel Optane's capabilities in serving as main memory device, persistent storage, and byte-addressable memory device to applications running in user-space. [83]shows the importance of NUMA-aware memory allocation at the application level and avoiding kernel overheads for Optane PMM as poor applications of both concepts are more expensive on Optane PMM than on DRAM. [84] explores into using Optane persistence memory on virtual machines to demonstrate that the needed size of DRAM is very small. The combination of DRAM and high bandwidth memory (HBM) in Intel Knights Landing platform is discussed in [81]. The basic ideal of these works is to achieve a high application performance with introducing fast memory as much as possible.

Use page as the basic unit for data management is proposed in [22] [18] [85] [21]. These work prefer to profile memory access behaviors to decide where a page should be place. Thermostat [22] only profiles the access information on 0.5% of the total memory pages instead profiling all pages. Because the whole page profiling could result in a 4x performance slowdown. Similarly, Heteroos [18] tracks and collects information of hot pages by setting and resetting PTE, which brings a very high overhead due to it triggers too many context switches and data transfer operations. Limiting the amount of pages to profile is a possible solution to reduce overhead. For example, [85] and [21] introduce a specific hardware to periodically collect the access information on main memory. [85] and [21] are

kind of lightweight, but they can incorrectly calculate the total times of memory accesses on short-lived data objects due to the default performance loss in sampling.

Implementing transparent unified memory access between CPUs and GPUs has also been well studied in the past works. vDNN [31] proposed a heterogeneous memory system between CPUs and GPUs by leveraging the structure of the training computation graph used by DNN is static and follows a sold patter. Moreover, the intermediate tensors generated in the earlier computation step may not be used until some later computations are performed.SuperNeurons [29] introduces a dynamic GPU memory scheduling runtime to enable the network training far beyond the GPU DRAM capacity and a cost awareness algorithm in applying re-computation of forward pass layers in the backward pass in training DNN models to reduce memory. moDNN [30] adopts the idea of data offloading and prefetching which proposes a novel sub-batch size selection method which cooperates with data transfer scheduling to further reduce memory usage without impacting the accuracy and ensures the memory usage tightly fits the memory budget.

3.4 Memory Management Strategies on Unified Hybrid Memory System

An extensive study on data management strategy in NUMA-based systems has been proposed, such as HMvisor [19], Nap [20], HiMUMA [36] and Carrefour [34]. HMvisor [19] is designed for dynamic hybrid memory management for virtual machines. It proposes a lightweight page migration strategy by decoupling page hotness tracking from page migration. HMvisor [19] also propose a memory resource trading policy to adjust the capacity of DRAM and NVM for each VM, with the monetary cost unchanged. Nap [20] implements a black-box approach to convert concurrent persistent memory indexes into NUMA-aware counterparts. It introduces a NUMA-aware layer (NAL) on the top of existing concurrent PM indexes, and steers accesses to hot items to this layer to tackle skewed data access problems. HiMUMA [36] is specifically develpoed for data management in NUMA-bases systems. It proposes hybrid memory management strategy based on data hotness. HiMUMA also optimizes the balancing algorithm used in NUMA with consideration the bandwidth

differences so as to guarantee a higher memory bandwidth utilization. Carrefour [34] introduces a hardware-based memory management to reduce traffic congestions in NUMA systems. Those previous work focus on memory management strategy for universal workloads running in NUMA systems. However, in a NUMA-based system equipped with hybrid memory hardwares, the heterogenous memory characteristics in access and latency is further amplified, especially when deep learning applications are running on it. UniRedl is designed to manage data migration for big data and deep learning applications. We propose a data access pattern and computation graph awareness strategy, and memory bandwidth utilization-based algorithm to maximize the application performance.

There are also some works explore page placement strategies in hybrid memory systems BMPM [86], OIM [21], Thermostat [22], HeteroOS [18]. BMPM [86] proposes bandwidth-aware memory placement and migration policies to solve the problem caused by the bandwidth difference of the memory nodes in a heterogeneous memory system. But its solution is still based on the traditional automatic NUMA balancing algorithm, which may not be efficient for big data and deep learning applications. OIM [21] propose a page placement similar to a widely used page replacement strategy used by Linux. It improves page migration performance by launching a couple of threaded to migrate the single pages and concurrent migration to handle multiple pages. However, adopting this strategy to determine the page migration for deep learning applications with various data access pattern can be slow and has no clues on the global view. Thermostat [22] proposes an application-transparent huge-page-aware mechanism to place pages in a dual-technology hybrid memory system. It uses an online page classification mechanism to classify both 4KB and 2MB pages as hot or cold while incurring no observable performance overhead across several representative cloud applications. HeteroOS [18] proposes an application-transparent OS-level solution for managing memory heterogeneity in virtualized system. It extracts rich OS-level information about applicationsâ memory usage to place data in the suitable memory to avoid page migrations. However, this memory management strategy

is not mainly designed for NUMA-based systems, and fails to achieve the optimal performance improvement. UniRedl exploits the inherent feature differences in hybrid memories and the specific computation characteristics in deep learning applications to improve the efficiency of using hybrid memories and the performance of applications.

As discussed in Section 1, implementing Hybrid Memory System for In-memory computing involving multiple objectives and challenges. There is a significant body of work in this area that propose various solutions with promising performance improvement, but there are critical limitations on the effectiveness and efficiency of existing techniques, as discussed in Section 2. We develop a hybrid memory system that includes fast and relatively slow memory hardware and memory management strategies for applications running in cloud environments based on optimization formulations, feedback control, and deep learning methods.

CHAPTER 4: A SHARED MEMORY CACHE LAYER ACROSS MULTIPLE EXECUTORS IN APACHE SPARK

4.1 iMlayer System Design

Unlike the default memory management strategy in Apache Spark, iMlayer allows multiple executors on a single node to share a specific off-heap memory space with each other. It aims to improve the hit rate of intermediate data blocks by caching and evicting data uniformly across multiple executors on the same hosting machine. The key insight of iMlayer is to develop a new eviction strategy applied in the proposed shared cache memory space. Although there are many different eviction strategies [87] [88] on cache management, they cannot guarantee a good performance while multiple executors are co hosted on slave machines in Apache Spark. To achieve memory hit rate, we propose a metric named Next Re-reference Distance (NRD) to predict how far a data block will be referred again, and this value gets updated once a referring request coming. When memory tensions occur in cache space, there are some cached data blocks to be evicted to make free space for upcoming data. Under this situation, iMlayer would take all cached blocks into consideration to choose the ones that have little possibilities in re-referred (whose NRD is bigger) to ensure a higher hit rate.

Figure 4.1 shows the overall architecture of iMlayer, which consists of three major components, i.e., $iMCache$, $iManager$ and $iMonitor$.

- $iMCache$ is a cache memory space donated from individual executors and replaces the isolated off-heap memory region exclusive to each executor in vanilla Spark architecture. It is responsible for recording the reference information of every data block, i.e., NRD, which denotes blocks' re-referring possibilities in the future. Then

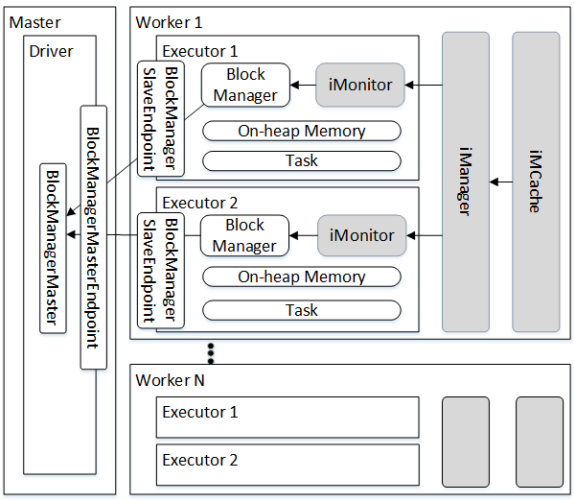


Figure 4.1: System architecture of iMlayer.

NRD is used to decide which block should be evicted when the memory tension occurs.

- $iManager$ is responsible for managing the data blocks cached in iMCache with a unified caching and evicting operation. By leveraging global data referring information, it evicts less possibly used data and makes more free space for the coming blocks in order to guarantee higher overall hit rate.

- $iMonitor$ is running on each executor and responsible to maintain the block information belongs to individual executor, e.g., owner's executorID, storage location and reference statistics. It periodically reports the data block's location in iMCache to the original $BlockManager$.

4.2 Memory Management in iMlayer

4.2.1 Memory sharing policy in iMCache

Figure 4.2 depicts that iMlayer integrates these isolated executors via sharing the off-heap memory spaces from multiple executors. By default, each Spark executor is allocated with exclusive on-heap and off-heap memory space respectively. LRU policy is applied to manage the data eviction in isolation. When multiple executors (e.g., Executor #1 and

#2 in Figure 4.2) are deployed on a single node, the computational behaviors of different executors may not be identical so that the data access demand can vary a lot over time. For example, the left two graphs show data reference tendency, where x-axis represents timeline and y-axis depicts dynamic data block access rates. In this case, such isolated memory management of individual executors could not always provide a good performance due to its individual management strategy.

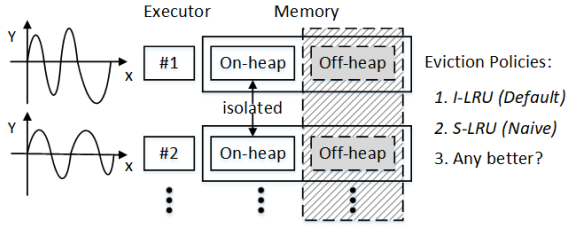


Figure 4.2: Memory sharing among multiple executors.

To tackle this problem, we separate the original on-heap and off-heap memory space, and then combine off-heap memory segments from different executors together as a unified cache space (i.e., iMCache as shown in Figure 4.2), so that executors can share iMCache with each other. Then iMCache has to employ an efficient cache management policy to guarantee the flexibility of memory usage with multiple executors. Intuitively, we apply the default LRU strategy in the shared iMCache, which is denoted as S-LRU strategy. I-LRU (i.e., default Isolated LRU) is used to distinguish with the LRU policy used in on-heap memory without shared off-heap memory space. Figure 4.3(a) provides the system performance comparison between I-LRU and S-LRU approaches while running Kmeans workload. It is obvious that S-LRU outperforms I-LRU 30% on average in terms of data hit rate. However, we also find naive S-LRU cannot always perform well at runtime, such as the period between 12 to 14 seconds. The above observation confirms the potential benefits of sharing off-heap memory among executors, and motivates us to develop a more efficient cache management strategy. However, there are significant major challenges when applying iMCache: (1) how to efficiently allocate shared cache memory among multiple executors? (2) when memory tension occurs in this region, how to select data blocks to be

(a) I-LRU vs S-LRU (b) Runtime with Sizes

Figure 4.3: Figure (a) shows the comparison between I-LRU (Isolated LRU) and S-LRU (Shared LRU) eviction policy. Figure (b) shows the runtime impact with different shared off-heap memory sizes.

evicted?

4.2.2 Memory allocation of iMCache

There are various works [89] [90] on elaborating cache size configuration to tackle some of the above problems. However, these existing methods are not flexible and applicable on Apache Spark due to the unique in-memory computing characteristic [1]. Thus, we conduct a case study to investigate the sensitive cache size setting while sharing off-heap memory. Figure 4.3(b) depicts the job runtime achieved under different shared off-heap memory size configurations with two executors deployment model. It is obvious that the runtime decreases with the growth of the off-heap memory at the beginning stage, and the best performance comes with 256MB configuration. However, the performance dropped when memory size is over 256MB, reason behind this phenomenon laid on the fact that larger memory comes with more GC activities [91] leading to a longer runtime. This observation demonstrates a reasonable amount of shared memory can benefit the application performance while oversharing memory may incur performance degradation. Given this fact we focus on designing an efficient memory sharing policy to build iMCache as follows.

$$y_i(t) = R_i(r(t), e_m, s(t), \xi(t)).$$

This formula describes the relationship between input variables and output variable. The input variables are as following: $r(t)$ is the total memory allocation, e_m represents the number of executors deployed on a single node, the memory size of each executor should

denote is expressed as $s(t)$ and the regression vector is $\xi(t)$. The output $y_i(t)$ is the average task completion time for each job. As many Spark applications have predictable structure in terms of computation and communication, iMlayer predicts the $s(t)$ for a specific job based on monitoring the similar job's previous runs [92]. Here, the regression vector $\xi(t)$ contains a series of lagged outputs and inputs from the previous sample intervals, which is expressed as

$$\xi(t) = [(y(t-1), y(t-2), ..., y(t-n_y)),$$
$$(r(t), r(t-1), ..., r(t-n_r))]^T.$$

where n_y and n_r represent the number of lagged values for outputs and inputs. Let ρ denotes the number of elements in $\xi(t)$, so we can have

$$\rho = n_y + n_r.$$

R is the rule-based fuzzy model that consists of Takagi-Sugeno rules [93]. R_i, a rule of *R* which means a relation between application's allocated memory and its running time. Moreover it can be represented as

$$R_i\text{: IF } \xi_1(t) \text{ is } \Omega_{i,1}, \xi_2(t) \text{ is } \Omega_{i,2}\text{,..., and } \xi_\rho(t) \text{ is } \Omega_{i,\rho}$$
$$r(t) \text{ is } \Omega_{i,\rho+1} \text{ and } e_i \text{ is } \Omega_{i,\rho+2}, s(t) \text{ is } \Omega_{i,\rho+3},$$
$$\text{THEN } y_i(t) = \zeta_i \xi(t) + \eta_i r(t) + \omega_i e_i + \delta_i s(t) + \theta_i.$$

In this formula, Ω_i is the antecedent variables of the ith fuzzy rule, which is consisted of $\Omega_{i,1}$, $\Omega_{i,2}$, ..., $\Omega_{i,\rho+1}$, $\Omega_{i,\rho+2}$, and $\Omega_{i,\rho+3}$. The value of parameters ζ_i, η_i, ω_i, δ_i and offset θ_i are calculated by offline training based on application's running logs. Furthermore, each fuzzy rule describes a nonlinear relationship between the shared memory size and the corresponding task completion time for a specific workload.

To obtain the above model, some necessary parameters, i.e., $y(t), r(t), e_m, s(t)$ are first parsed from the workload's historical logs. A pattern model is built (as shown in Line 3-6 in Algorithm 1) to describe the relationship among those parameters following the

Algorithm 1 Cache Donation

Input: app_type, app_log
Output: $iMCache$: memory size in terms of $s(t)$
1: // Building Fuzzy Model
2: **function** CALOPTMEM(app_type, app_log)
3: // Abstract parameters $y(t), r(t), e_m, s(t)$
4: mem_para = parse(app_log)
5: //Build Fuzzy model
6: R_i = bldfuzzy(mem_para)
7: // Get $s(t)$ of each executor running specific workload based on relation model R
8: $s(t)$ = getSharedMem(app_type, R)
9: set $iMCache$ according to $s(t)$
10: set $cache\_mem$ = allocMem($s(t)$)
11: **end function**

proposed fuzzy model. In particular, different workloads may follow various patterns so that we use R to maintain all possible relation models. When a workload is deployed, our architecture decides the memory size that each executor should denote based on its type and performance model. After getting the parameter $s(t)$ for each executor, the size of iMCache is determined, which is shown by Line 4-10 in Algorithm 1.

4.2.3 Eviction Policy in iManager

Given the fact that the memory reference behaviors of Spark applications vary dramatically by different stages, jobs and applications, the naive cache eviction policy (i.e., LRU) may not guarantee a good and stable performance. As the shared cache memory space is typically much larger than the default exclusive memory of each executor, we propose a new eviction policy based on next re-reference distance, which concept is adopted in [58] [59].

1. *Next Re-reference Distance (NRD)*: We first introduce a metric, i.e., Next Re-reference Distance (NRD), to predict the possibility of a cache data block to be referenced again. Moreover, M-bit per cache block is used to denote one of its 2^M possible Next Re-reference Distance. NRD of each data block cached in iMCache dynamically gets updated once a block reference operation is requested. An NRD of zero implies that a cache block is

predicted to be re-referenced in the near future while NRD of saturation (i.e., 2^m-1) means that a cache block is supposed to be re-referenced in a longer future. Quantitatively, data blocks with small NRDs are supposed to be re-referenced sooner than blocks with larger NRDs.

The key role of NRD is to prevent blocks with longer re-reference distance from occupying the limited cache space too long time. Without any historical or external block reference information, NRD of each block is calculated by statical prediction. Since always assigning a 0 or 2^m NRD to newly inserted data block could not guarantee robustness across all block reference sequence. If the newly inserted data block is assigned with a 0 NRD, its re-reference distance will be updated so frequently that NRD fails to describe the real re-reference sequence. Oppositely, set newly data block's NRD to be 2^m causing longer cache occupation. So we assign the NRD of newly inserted data block to be 2^m-1, which value could guarantee the freshness of data blocks in cache. Additionally, always assigning 2^m-1 instead if 2^m brings more time to learn and improve the re-reference distance prediction.

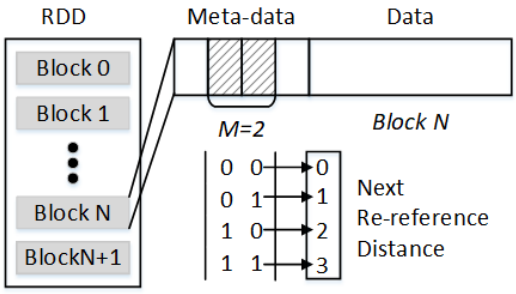


Figure 4.4: NRD based data structure of an RDD block.

An eviction strategy in terms of NRD is implemented in iMCache. Figure 4.4 depicts an example of the data structure of an NRD based RDD. Each RDD contains several data blocks. A data block is highlighted to illustrate its components. For each data block shown in this example, it mainly has two parts: *Meta-data* is used to describe attributes of data block while *Data* contains the real contents of each block. In particular, the shadowed section belongs to the meta-data part is a 2-bit (*M*=2) marker. We use this section to denote

four ($2^2 = 4$) possible situations of NRD (0, 1, 2 and 3). As is mentioned above, the NRD of each newly inserted data block is set to be *3*, which is denoted as *11* in the marker section of *Meta-data* part. Its metric gets decreased if this block is just re-referred. When cache space is full, data blocks with large NRD (*3* in this example) is the victim to be evicted while blocks whose NRD being *0* are highly kept in the cache space.

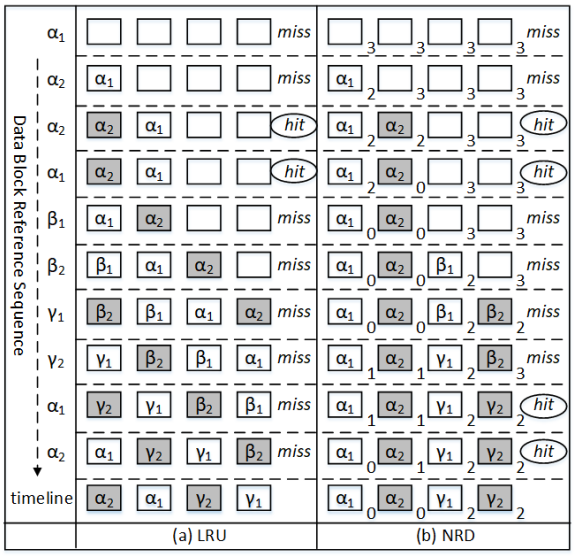


Figure 4.5: Behavior of LRU and NRD in reference sequence.

Figure 4.5 provides an intuitive behavior comparison between our proposed NRD policy and the default LRU method in the case of shared cache. In this example, we deployed two executors on a machine. To distinguish individual blocks belong to each executor, we use white blocks to represent Executor #1's and shadowed ones are for Executor # 2. Besides, the shared cache memory size among executors is set to be 4. So M=2 bits is used to represent the predicted re-reference distance for each block. Besides, this metric is denoted by the subscript outside each block in the right part of Figure 4.5. We use the following mixed block reference sequence from two executors to illustrate cache replacement behavior in shared cache space: $\{\alpha_1, \alpha_2, \alpha_2, \alpha_1, \beta_1, \beta_2, \gamma_1, \gamma_2, \alpha_1, \alpha_2, ...\}$.

The left column represents the behavior of LRU while our policy's behavior is provided in the right column. At the very beginning, two policies display a similar performance that both of them bring more missing than hitting references. However, as reference moving

on, NRD policy outperforms the default LRU by more hitting references. We observe that NRD gets 4 cache hits and outperforms the original LRU with only 2 cache hits. The example shows that our policy provides a better hitting performance by correctly predicting a shorter next re-reference distance for the most possible re-used cache blocks and a longer next re-reference distance for blocks that would not soon be referred again. In this case, the hit rate is significantly increased by NRD compared with the default LRU policy.

2. *Eviction Strategy*: Next Re-reference Distance (NRD) replacement policy is a modified LRU eviction strategy. It uses NRD to predict a cached block should be kept or evicted when the next block reference request comes. The primary goal of NRD is to prevent blocks with a shorter re-reference distance from being evicted. Without any external re-reference information, NRD statically predicts each block's re-reference distance. To make eviction more robust, NRD always inserts new blocks with a longer re-reference distance. 2^M-1 is applied to denote a longer re-reference distance. The intuition behind always assuming a longer re-reference distance on cache insertion is to prevent cache blocks with re-reference in the future from polluting the cache.

Algorithm 2 Block Eviction

Input: $cache\_mem$, $refer\_sequence$
Output: Blocks get evicted

```
1:  function EVICTBLOCK(cache_mem, refer_sequence)
2:      for (block b in refer_sequence) do
3:          if (b got hit in cache_mem)
4:              set r of block b to '0'     //r is NRD
5:          else search 1st block with r = 2^M − 1 & update
6:              if (found this block)
7:                  new block cached & set its r = 2^M − 2
8:              else increase all blocks' r by 1 & repeat Line 6
9:      end for
10:     CacheMonitor.updateInfo(evictedBlocks)
11: end function
```

Algorithm 2 shows the procedure of NRD eviction policy. Two parameters are used as the inputs for EVICTBLOCK: $cache\_mem$ is the iMCache space shared among executors and $refer\_sequence$ maintains a mixed block reference sequence from all execu-

tors. When reference sequence ($refer\_sequence$) is not empty and new data referring request aNRDes, the algorithm first checks if the demanded data block is already cached in $cache\_mem$. If so, its NRD is set to be 0 (a shorter interval), which denotes this block is just referred and has a high possibility to be re-referred in the near future.

In the case of cache miss, a victim block is supposed to be selected by finding the first block to be re-referenced in the distant future. This victim block will be evicted to make space for the coming block. We define the block whose NRD is $2^M - 1$ as the victim. The list parameter $evictedBlocks$ who maintains information on evicted blocks is updated by adding the victim's information into it. This strategy breaks ties by always starting the victim search from a fixed location. There are possibilities that victim selection failed as a block with a long distant re-reference distance can not be found. In this case, our method updates NRDs of all cached blocks in $cache\_mem$ by incrementing 1 and repeats the search until a block with a distant re-reference distance is found (Line 6-8). Then the information maintain in $evictedBlocks$ is updated. Finally, the information maintained by $evictedBlocks$ is sent to iMonitor component (Line 10), whose duty is maintaining blocks' basic information to help Block Manager locate demanded blocks rapidly and timely. EVICTBLOCK would not be invoked when there is no block request coming or memory tension occurs.

4.3 iMlayer Implementation

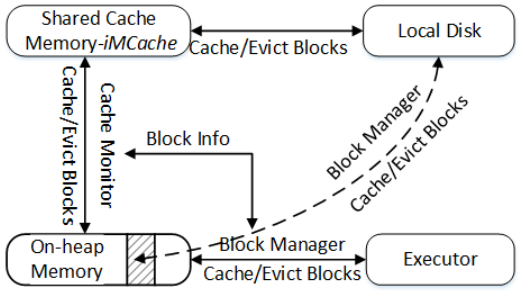


Figure 4.6: Workflow comparison between iMlayer and vanilla Spark.

$iMlayer$ is implemented with three major modules, i.e., iMCache, iManager and iMon-

itor, which are written in Java on top of Spark 2.2. Figure 4.6 shows the comparison of data interactive workflow between iMlayer and vanilla Spark. In Spark, the default workflow mainly consists of three parts: Executor, On-heap Memory and Local Disk. When Spark application begins to run in an executor, the on-heap memory space designated to this executor is empty. As the computation moving on, new data blocks would be cached in this space. As shown by solid rectangles and lines, we added iMCache between executor's on-heap memory and local disk. To uniformly manage the cached data block, we disable functionalities of $StorageMemory$ region in on-heap by configuring its size to be 0. So the intermediate data generated by all executors would be cached in iMCache. Caching and evicting operations in iMCache are executed by iManager under the control of iMonitor.

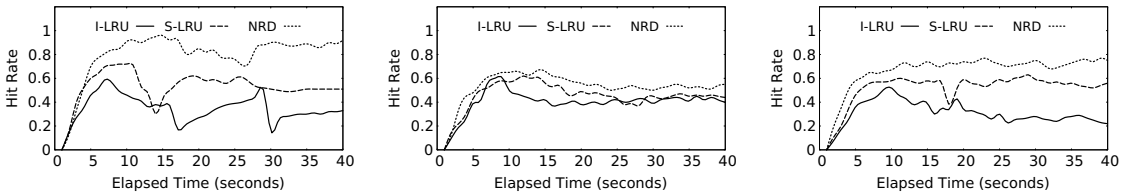
iMCache uses a process $allocate\_mem$ to initialize sharing layer where cached blocks are uniformly managed. rec_refer_info is the function to record cached blocks reference information and which executor it belongs to. Referring information gets updated once any reference request submitted. When an executor finished its job, $free\_cache$ is called to release iMCache space it took. By this operation, free space is available to cache blocks for other running executor sharing this memory layer. **iManager** is responsible for managing data blocks cached in iMCache. To implement this function, two core operations are applied in this component: CACHE and EVICT. CACHE operation handles the request of storing blocks evicted from executor's on-heap memory space. A list parameter $refer\_sequence$ is adapted to maintain the mixed reference requests from all executors. EVICT is used to decide which blocks cached in $iMCahce$ should be evicted when memory tension occurred. Our proposed eviction policy NRD is used in EVICT. The function $EvictBlock$ implements key operations of EVICT by choosing blocks with higher NRD and updates blocks information in $evcitedBlocks$. Furthermore, when evicted blocks were successfully placed into local disk, their location information is then maintained by $disk\_addr$ function. **iMonitor** maintains the block information between $BlockManager$ and iManager. When any CACHE or EVICT operations executed,

the monitor process keeps the updated information in meta-data section and conveys to $BlockManager$ via $updateInfo$ function. If executor fails to find the requested block in its on-heap memory, it will ask $BlockManager$ whether those blocks existed in iMCache with function $read\_imcache$. If this block gets hit, memory address in iMCache will be provided and then corresponding blocks would be fetched directly. However, when those blocks are not cached, iManager then tries to search them on the local disk by calling function $search\_disk$.

4.4 Evaluation

4.4.1 Experiment Setup

The evaluation testbed consists of 9 nodes, each of which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR4 RAM, running Ubuntu 16.04 LTS operating system with kernel version 4.0, Scala 2.10.0, and Hadoop YARN 2.8.0 for cluster management. One node serves as the *master*, and all the other 8 nodes serve as *slaves*. These nodes are connected with Gigabit Ethernet. We apply the default resources management policy and Fair Scheduler strategy to control resources (CPU & memory) allocation and job scheduling. To test our prototype, we use three workloads from the HiBench big data benchmarking suite [45]. We run each workload 5 times to get an average performance so that accuracy could be guaranteed. We compare our method with the following two representative approaches. (1) I-LRU (Isolated Least Recently Used): the default management policy adopted by Spark, and merely evicting blocks based on the last time they are referenced. (2) S-LRU (Shared Least Recently Used): we introduce the iMCache and use LRU in this shared memory space. In contrast, our proposed eviction policy (i.e., NRD) that takes NRD into consideration when managing cached blocks. Unless otherwise noted, all experiments were conducted on resource allocation of Case #2: two executors are deployed on the same worker node, each executor is allocated with 4 CPU cores and 4GB exclusive memory.



(a) Kmeans (b) SVD (c) PageRank

Figure 4.7: Hit rate traces under the three eviction policies with different workloads.

4.4.2 Improvement on Block Hit Rate

Figure 4.7 depicts the hit rate variation tendency within a typical time period (0s-40s) at runtime. The results show that hit rates from different architectures keep increasing at the beginning stage due to more requested data blocks cached in memory. However, hit rate decreases occasionally in later stages when the cache memory space is full and cache miss occurs. In this case, the eviction policy should evict several unnecessary blocks in order to load required ones, which explain the rise of hit rate in each curve. Figure 4.7 further demonstrates that iMlayer outperforms vanilla Spark by achieving higher hit rate and decreasing fluctuations in workloads' tendency. Hit rate changes of Kmeans and PageRank (plotted in Figure 4.7(a) and (c) respectively) are more obvious than SVD's changes (shown in Figure 4.7(b)).

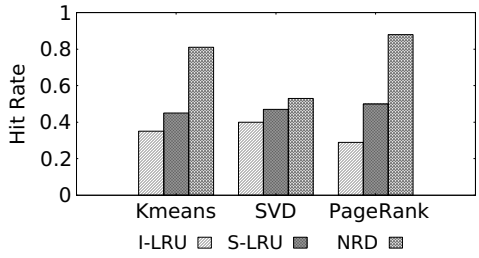


Figure 4.8: Average hit rates in different cases.

Figure 4.8 compares the average hit rates achieved by different cases. iMlayer with NRD obtains average 45%, 16% and 27% improvements respectively on these workloads compared to the vanilla Spark with default LRU. The results also demonstrate that iMlayer with LRU achieves 19%, 5% and 6% enhancements compared to the vanilla Spark architecture.

A shared cache layer can effectively improve cached blocks hit rate by caching and evicting blocks uniformly, as it takes all executors' demands into consideration when evicting cached blocks. Moreover, the hit rate improvements of Kmeans and PageRank from iMlayer are more obvious than SVD. Reason behind this phenomenon is the fact that SVD is a CPU-bound workload and the benefit of memory utilization improvement is limited. In most cases, the assigned memory space is sufficient to cache SVD's demanded data blocks.

4.4.3 Improvement on Reference Locality

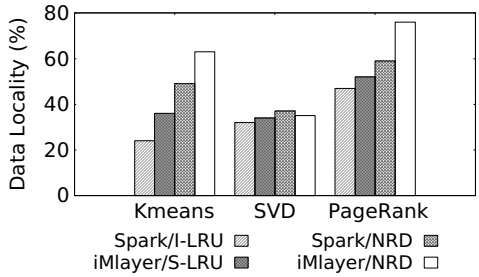
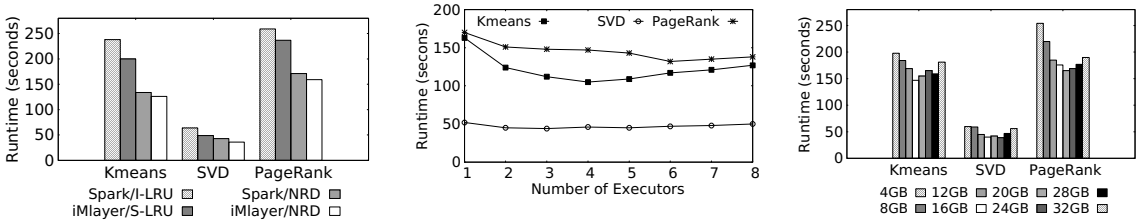


Figure 4.9: Data reference locality.

Data Locality indicates how much valid or useful data are preserved in cache space. With higher data locality percentages, frequency of data movement in cache is decreased. Figure 4.9 investigates performance on data locality with four cases. Comparing the whole results, iMlayer outperforms vanilla from the view of architecture and NRD defeats LRU in the point of eviction policy. iMlayer with NRD obtains the best performance by bringing 62%, 8% and 38% improvements compared to vanilla Spark with LRU for Kmeans, SVD and PageRank respectively. NRD alone still obtains about 16%, 13% in average data locality when comparing Spark/I-LRU v.s. Spark/NRD and iMlayer/S-LRU v.s. iMlayer/NRD. Data locality percentages in Kmeans and PageRank increases obviously with the iMlayer (a larger shared cache space) or NRD policy (more accurate prediction in block preserving). However, the performance of SVD's is relatively stable under the different four cases because the reference behavior of SVD does not change too much and the allocated cache space is large enough to preserve data blocks it demands.



(a) Impact of Eviction Policies (b) Impact of Executors (c) Impact of iMCache Sizes

Figure 4.10: Performance impacts under different eviction policies, number of executors and iMCache sizes.

4.4.4 Improvement on Job Runtime

Figure 4.10(a) shows job runtimes of three chosen workloads under various cases. In general, our work deployed with NRD achieves the best performance by obtaining 47%, 43% and 38% improvements compared to vanilla Spark with LRU for all above workloads. Particularly, vanilla Spark deployed with NRD outperforms it adopts LRU (Spark/I-LRU) with about 43%, 32% and 33% improvements respectively. It demonstrates that NRD eviction policy can dramatically improve the job level performance by efficiently managing the data blocks in limited memory space. When applying naive LRU policy in iMCache, it (i.e., S-LRU) outperforms vanilla Spark with LRU about 16%, 23% and 9% in job runtime. It further demonstrates the memory sharing policy can effectively improve job level performance by uniformly managing blocks.

Figure 4.10(b) depicts various job runtimes under the iMlayer with different numbers of executors deployed on each slave node. The experimental results show Kmeans and PageRank benefit more than SVD when increasing the number of executors from 1 to 8. Figure 4.10(b) also illustrates that more executors may not always bring better performance. The best performance of Kmeans and Pagerank are achieved with 4 and 6 executors respectively. However, SVD's performance does not change too much under the different numbers of executors because the cache behavior of SVD (a CPU-bound workload) is relatively stable and does not need too much memory interactions.

Figure 4.10(c) investigates the performance impact of iMlayer with various iMcache

sizes. In this experiment, we setup 8 cases with various iMCache sizes range from 4GB to 32GB. Moreover, NRD is adopted for cache management. In Figure 4.10(c), three workloads display similar performance behaviors in terms of the cache size impact. When iMcache size is small (e.g., 4GB, 8GB and 12GB), workloads' runtime is longer because small memory size could not cache enough data blocks so that more frequent I/O operations occurred between memory and local disk. In particular, note that these job runtimes start to drop down if the shared memory sizes are over provisioned because of the fact that too large memory size may lead more GC operations.

4.4.5 Overhead Analysis

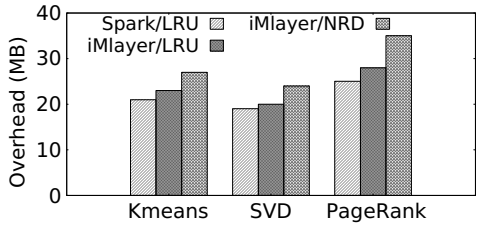


Figure 4.11: Memory overhead of vanilla Spark and iMlayer.

To analyze the overhead of iMlayer, we conduct experiments to measure the memory space consumptions in the eviction policy process while applying different approaches, i.e., vanilla Spark with LRU, iMlayer with LRU and iMlayer with NRD, respectively. All these relevant processes are running on java virtual machine. We assign 4GB memory resource to each executor, which follows the practice experiences by existing works [89]. Figure 4.11 demonstrates that the memory overheads caused by different eviction processes are far more less than typical executor's memory sizes (i.e., 1GB to 24GB [89]). The overheads of eviction processes deployed in iMlayer are a little higher than vanilla Spark with LRU. The reason is that more information about performance metrics have to be maintained for the shared cache management and NRD needs an extra memory space to record NRD of each data block. In particular, the average memory overheads caused by iMlayer with LRU and NRD are about 10% and 20% more than vanilla Spark with LRU respec-

tively, which are negligible to the whole system-level memory resource. Considering the performance improvement achieved by iMlayer, such memory overhead is an acceptable trade-off between space and time consumption.

CHAPTER 5: DATA LIFE AWARE MODEL UPDATING STRATEGY FOR STREAM-BASED ONLINE DEEP LEARNING

5.1 iDlaLayer Architecture Overview

We present iDlaLayer, a thin middleware layer between applications and backend deep learning frameworks that realizes efficient online model updating.

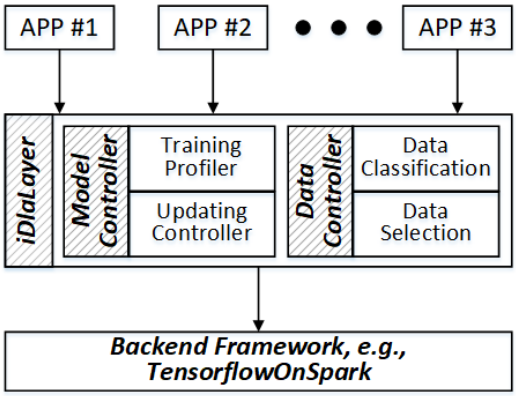


Figure 5.1: Architecture of iDlaLayer.

Figure 5.1 provides an overview of iDlaLayer architecture. The critical contribution relies on a novel Data Life Aware model updating strategy (DLA) to online learning applications. DLA is capable of building better training data samples in terms of contribution made by data from different life stages and determining the right time to perform model updating with a lower training cost. More details on updating strategy and data life cycle would be discussed below. iDlaLayer mainly consists of two components as shown in Figure 5.1: **Model Controller** and **Data Controller**.

- **Model Controller** has two functional parts. The *training profiler* logs and profiles the training time for each application. We use linear regression approach [94] [95] to predict time consumed to train a model based on the size of sample data. The

prediction result is then used as an estimation of future arrival data. The *updating controller* is responsible for deciding when to perform model updating and take the role of communicating with the data management component. Besides, it is the module where we implement DLA.

- **Data Controller** maintains and builds training data samples in terms of life stages. This component communicates with the updating controller to evaluate the contribution each data made to model updating and then use such information as an index determining data life stage among *Newborn*, *Mature*, *Preserve* and *Discard*. Moreover, the training data sample is built in this component based on the contributions to model updating made by data from various life stages.

For centralized deployment, all applications interact with iDlaLayer through a client exclusive to each other. The applications are also associated with backend frameworks, e.g., TensorflowOnSpark, to implement their training logics. iDlaLayer communicates with backend for model training and sample data management through RPC connection. This architecture design implements a versatility feature while scalability and efficiency are achieved with only one iDlaLayer thread deployed in backend frameworks.

For distributed deployment (including a master node and several slave nodes), we deploy iDlaLayer in the master node, which is responsible for data management, and communicate with all slaves to perform model training. When iDlaLayer is used to manage multiple applications, the application-specific information maintained by model controller and data controller, e.g., the mount of arrival data, last model update time, data life stages, etc. are necessary. A possible solution to organize this information is to launch an exclusive iDlaLayer thread for each application and manage information individually. However, multiple threads always bring too much context switches overhead. To release system pressure, we only deploy a single iDlaLayer thread in the master node to maintain the information of all applications and allow each application to communicate with iDlaLayer asynchronously.

5.2 Data Controller

5.2.1 Data Life Cycle

Before describing the general training workflow, we first introduce a concept named **Data Life Cycle**. The data life cycle is a sequence of stages that a data sample goes through from its initial generation to its eventual discarded at the end of its life. In this paper, we mainly divide the data cycle of data into four stages: *Newborn*, *Mature*, *Preserve* and *Discard* according to its performance in updating model quality.

- **Newborn**: In this stage, data arrivals from various sources such as users, databases and previous processing stage. Typically, data staying in this stage is ready for being incorporated for model training and this waiting time may vary a lot due to the current model updating strategy.
- **Mature**: Sample data gets its turn to be absorbed by the current model training process in this stage. The contribution to improve the training model is supposed to be evaluated, which is used to decide whether this data should be preserved for future usage or be abandoned directly.
- **Preserve**: The data contributes a lot in improving model training and is preserved for future usage. Given the fact that online learning has no knowledge of future coming data, preserving some data samples is helpful in training a more accurate model and saving model training time.
- **Discard**: This is the last stage, in which the data is discarded because it cannot make any contributions to model updating. Meanwhile, other data samples with better performance may be available. Keep discarding and updating the preserved data could benefit model updating that lives up to the changing environments.

Figure 5.2 depicts an example of a general dataflow in iDlaLayer. From left to right, it is a timeline following the application's running. Two trainings are introduced to demonstrate

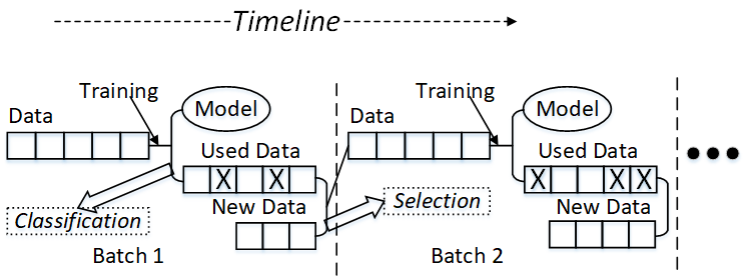


Figure 5.2: Dataflow in iDlaLayer.

the online learning loop. Some data samples (illustrated as "Data" in Figure 5.2) are used to train the model for each batch. When training model finished, an updated model is built and the training data is labeled as "Used Data" samples because life stages of data it includes have changed. We denote this procedure as *Data Classification* and illustrated in Figure 5.2. Any individual data in *Discard* stage is marked with the "X" symbol, which means this data is supposed to be replaced by new arrival data in the next training data sample. During the training model with data samples, some new data also arrived and are ready to be incorporated for subsequent training. iDlaLayer then merges new data into "Used Data" samples by abandoning data in *Discard* stage according to a combination strategy. And then a new data sample is built by *Data Selection* procedure.

Moreover, our architecture is capable of deciding a better time of triggering model updating in terms of training cost. After obtaining those information, another model training starts, which is displayed as Batch 2 in Figure 5.2. Given the fact that numbers of arrival data vary in different training batch, we just directly incorporate all the new data into data samples to train model aiming to guarantee the freshness of data sample and the quality of the updated model. More details about data combination and model updating strategies are discussed in the next section.

5.2.2 Data Classification

Considering data are always generating and flooding into the application, it is difficult to evaluate the performance of training data. We introduce the metric *Data Contribution* to di-

rectly measure how much each data made to the quality of the training model. Specifically, *Data Contribution* is evaluated by the loss value from the training model with batched input data. Then we further divide the training data into different life stages in terms of their contribution to model qualities. The built-in evaluation method *model.evaluate()* provided by TensorFlow is adopted to assess the contribution of each data as well as the quality of the training model. Since we use the same training data to train and evaluate models, it is inevitable to bring some deviations to data contribution. To tackle this problem, we also consider data occurrence when estimating data contribution.

After obtaining data contribution, we adopt the *Fuzzy Model* to guarantee a better life stage division for different applications. In the fuzzy model, we use multiple inputs and single output (MISO) to evaluate the contribution each data made. It is based on the relationship investigation of data occurrence, model quality and application types. The fuzzy model is often used to capture the complex relationship between resource allocation and a job's fine-grained execution progress [96]. Given the periodic and repeatable feature of online learning applications, we design a fuzzy model based on historical running logs. We formulate a fuzzy model:

$$q_i(t) = R_i(f(t), c, s(t), \xi(t)).$$

In this formula, the input variables are as following: $f(t)$ is the occurrence frequency of data, c represents the contribution of each data, the data stage is expressed as $s(t)$ and the regression vector is $\xi(t)$. The output $q_i(t)$ is the quality of training model. As many online learning applications have a predictable structure in terms of computation, our architecture predicts the c for a specific application based on monitoring its similar previous runs. Here, the regression vector $\xi(t)$ contains a series of lagged outputs and inputs fof the prior sample batches, which is expressed as

$$\begin{aligned}\xi(t) = [&(q(t-1), q(t-2), ..., q(t-n_y)),\\ &(f(t), f(t-1), ..., f(t-n_r))]^T.\end{aligned}$$

where n_q and n_f represent the number of lagged values for outputs and inputs. Let ρ denotes the number of elements in $\xi(t)$, so we can have

$$\rho = n_q + n_f.$$

R is the rule-based fuzzy model that consists of Takagi-Sugeno rules [93]. R_i, a rule of *R*, which means a relation between data life stage and model quality. Moreover, it can be represented as

$$R_i\text{: IF } \xi_1(t) \text{ is } \Omega_{i,1}, \xi_2(t) \text{ is } \Omega_{i,2}\text{,..., and } \xi_\rho(t) \text{ is } \Omega_{i,\rho}$$
$$r(t) \text{ is } \Omega_{i,\rho+1} \text{ and } e_i \text{ is } \Omega_{i,\rho+2}, s(t) \text{ is } \Omega_{i,\rho+3},$$
$$\text{THEN } q_i(t) = \zeta_i \xi(t) + \eta_i f(t) + \omega_i c_i + \delta_i s(t) + \theta_i.$$

In this formula, Ω_i is the antecedent variables of the ith fuzzy rule, which is consisted of $\Omega_{i,1}$, $\Omega_{i,2}$, ..., $\Omega_{i,\rho+1}$, $\Omega_{i,\rho+2}$, and $\Omega_{i,\rho+3}$. The value of parameters ζ_i, η_i, ω_i, δ_i and offset θ_i are calculated by previous training loop based on the application's running logs.

5.2.3 Data Selection

With the concept of data life stage, we formulate how to combine training data samples at each batch. The formulation of the data selection strategy is as follows. Let *m* data samples arrive in a sequence, which follows a time sequence a_1, a_2,..., a_m. Let's assume the *i*-th update begins at time t_i and takes p_i time to finish, so that I_i contains the data arrived after the (*i-1*)-th update starts and before the *i*-th:

$$I_i = \{k \mid t_{i-1} \leq a_k < t_i\}.$$

At first, as datasets arrived at the system, we define that all of them are in *newborn* stage. Let N_i denote a set of *newborn* data samples. Similarly, M_i, P_i and D_i represent *Mature*, *Preserve* and *Discard*, respectively. We use I_i to indicate a collection of data samples to be incorporated by the *i*-th update. I_i consists of data comes from N_i, M_i, P_i and D_i, which is expressed as:

$$I_i = \alpha N_i + \beta M_i + \gamma P_i + \delta D_i. \qquad (5.2.3.1)$$

Here α, β, γ and δ represent percentages of data from each life stage. We need to optimize parameters' combination to guarantee that I_i contributes more in model updating.

So, the cumulative time cost, denoted L_i, is computed as:

$$L_i = \sum_{k \in I_i} t_i + p_i - a_k, \qquad (5.2.3.2)$$

$$\text{s.t. } I_i \in \{I\}, t_i, p_i > 0, a_k \in I_i,$$

where I is the available data sample set. Summing up L_i from all updates, we can get the data incorporation time cost with well-combined data samples to model updating: $\sum_i L_i$.

5.3 Model Controller

5.3.1 Training Profiler

We declare another metric *Training Cost* to measure the cost spent in training model. *Training Profiler* is responsible for profiling this metric and serving this information to decide model updating times. Training cost is unavoidable when updating models, which is directly measured by the machine-time in public from commercial or cloud providers. To simplify the scenario, we assume that the training is finished on a single physical machine in the following discussions. Therefore, the training cost of each model updating is equal to its training time p_i.

Based on our observations in the case studies, the training cost is in proportion to the amount of data that the training is performed. To be more exact, the training cost follows a linear regression model against the data mount, which can be expressed as:

$$p_i = f(I_i) = \lambda I_i + \varepsilon, \qquad (5.3.1.1)$$

where λ and ε are parameters required by the linear regression model, which can be obtained or determined by historical model updating activities. To measure the cost Training cost is unavoidable when updating models, which is directly measured by the machine-time in public from commercial or cloud providers. To simplify the scenario, we assume that the training is finished on a single physical machine in the following discussions. Therefore, the training cost of each model updating is equal to its training time p_i.

5.3.2 Updating Controller

The controller determines when to perform model updating considering training cost and data life stages. The above metrics alone could not guarantee good model quality and low training cost, a well-designed model updating strategy is indispensable for our architecture. In this section, we propose a model updating strategy deployed in *Updating Controller* based on data from different life stages.

Data Life Aware Strategy: The goal of iDlaLayer is to find a model updating strategy that could maximize data contribution in terms of data life cycle and lower training cost when handling dynamic data streams. In particular, we implement DLA to guarantee accurate model training with considering various contributions offered in different data life stages and determine when to perform the model updating operation. We formulate the objective function with a knob parameter *w*, which means the training cost for each data sample. It expects the data incorporation latency to be decreased by *w*. In this problem, latency L_i and processing cost p_i are unified and should be optimized simultaneously, and it can be expressed as:

$$min_{t_i}(\sum_i L_i + wp_i).$$

The first subgoal of the strategy is to build better training data samples I_i in terms of the contributions they would make to the model quality. Then the data incorporation latency L_i is calculated following Equation 5.2.3.2. To simplify the problem, we calculate the suitable I_i with a greedy historical algorithm, and then with it to minimize the data incorporation latency L_i. Finally, the problem can be transferred as:

$$min_{t_i}(\sum_i(\sum_{k\in I_i} t_i + p_i - a_k) + wp_i), \qquad (5.3.2.1)$$

$$\text{s.t. } I_i \in \{I\}, a_k \in I_i, w, p_i > 0.$$

The second subgoal is to find good updating decisions t_i, i.e., the model updating time point, which comes with a higher data contribution without knowing future data arrival.

Model updating could be performed at any time point as long as new data arrives. For any single random data sample, it is straightforward to calculate a suitable updating time t_i with traversing all possible solutions. However, the consumptions to obtain all combined streaming data samples' updating time points is enormous, whose worst case is checking all potential solutions. Thus, Equation 5.3.2.1 is an NP-hard combinatorial optimization problem. We adopt a *competitive analysis* method (which has been widely applied by previous studies [97] [98]) to solve the problem in this work.

Algorithm 3 Data Life Aware Strategy

Input: a_1, a_2,..., a_n: arrival time of untrained data
α, β, γ and δ: parameters of data combination
λ and ε: parameters in runtime model
w: weight in minimization objective
Output: t_i: the suitable update decisions

1: **function** CALCONTRI(i) //
2: $a_i = \alpha N_i + \beta M_i + \gamma P_i + \delta D_i$
3: **return** updated a_i and α, β, γ and δ
4: **end function**
5: *Step 2 Cal Training Cost*
6: **function** CALLAT(i, j) //
7: $p = \lambda\ (j - i + 1) + \varepsilon$ // Estimate Rumtime
8: $e = p + a\,[j]$ // End Time of Training
9: **return** $\sum_{k=i}^{j} e - a\,[k]$
10: **end function**
11: *Step 3 Make Update Decision*
12: **function** CALUPDATE
13: l = CalLat(*1*, n)
14: **for all** $k \in \{2, ..., n-1\}$ **do** //
15: l' = CalLat(*1*, k) + CalLat(k+*1*, n)
16: **if** $l - l' > w\varepsilon$ **then** // Estimate regret
17: Get and output t_i
18: **end function**

Online Learning Algorithm: Algorithm 3 depicts the main procedure of DLA, mainly including *Build Data Sample*, *Calculate Training Cost* and *Make Update Decision*. (1) Data sample is combined at the beginning of model updating according to Equation 5.2.3.1. Function *CalContri()* (Line 1-4) builds combined data sample based on contributions made by data from different life stages. Contribution here is measured by the model quality with

historical data samples to ensure better data combinations for upcoming updating. Besides, this function is responsible to update life stage of each data after new data arrives or an updating finished. (2) When data samples are ready to be incorporated, DLA enters Step 2 by committing *CalLat* (Line 6-10) to calculate training cost of data based on Equation 5.2.3.2. Training cost consists of time spent in incorporation and processing with combined data sample, respectively. DLA is designed to provide faster data incorporation at a lower training cost. (3) Based on data samples and training cost, decision on whether and when to perform a model update is made with function *CalUpdate* (Line 12-18) in Step 3, which is the key procedure to solve objective function. Then update decisions t_i as shown in Equation 5.3.2.1 are transmitted to backend frameworks to commit model updating.

5.4 Workflow and Implementation of iDlaLayer

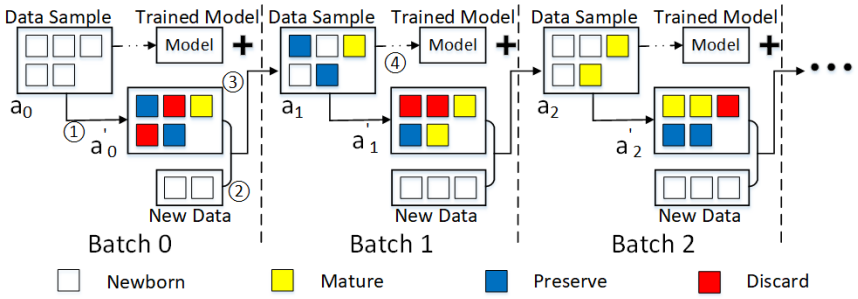


Figure 5.3: Detailed Training Workflow of iDlaLayer.

5.4.1 Workflow

Combing the concepts described above, Figure 5.3 depicts the detailed workflow of our proposed architecture. Three training model stages are included: $Batch0$, $Batch1$ and $Batch2$. Specifically, we adopt white, yellow, blue and red to denote data block in *Newborn*, *Mature*, *Preserve* and *Discard* stages respectively. Data samples containing several data blocks are used to train the model. $Batch0$ is the first model training with the data sample a_0 that includes 5 *Newborn* data blocks (illustrated with white blocks). Since data blocks contribute variously to model quality, their data life stages are different when model training finished, which is shown by different colored blocks in $a_0^{'}$. Step ① demonstrates

data life changes between a_0 and $a_0^{'}$. Comparing a_0 and $a_0^{'}$, red data blocks in *Discard* stage are replaced with new arrival data blocks and turns into $a_{i-1}^{'}$: 1 *Mature*, 2 *Preserve* and 2 *Discard* blocks. During the procedure of the model training with data sample a_0, some new data also arrived at the same time. If the current training finished and its training metrics are profiled, a new data sample a_1 (containing 2 *Newborn*, 1 *Mature* and 2 *Preserve* data blocks) for next training procedure $Batch1$ is supposed to be built (illustrated by Step ②). Reason behind this operation is that red blocks would contribute less for the coming model training, getting rid of those blocks brings more space and opportunities to new arrival data. Step ③ describes updating decision which is responsible for determining when to perform next training in terms of training cost. The next model training $Batch1$ is triggered with newly built a data sample and model from the previous training procedure. Furthermore, several performance metrics such as training cost, data contribution and "knob" parameter are profiled in Step ④. Similarly, $Batch1$ trains model and prepares data sample a_2 for next training stage $Batch2$. This training loop would continue until no more new data arrives or no improvement in model quality.

5.4.2 Implementation

We implement iDlaLayer with Java on top of TensorflowOnSpark. Figure 5.4 shows the workflow of iDlaLayer. (1) When a new data sample arrives, an application contacts its exclusive client process running in **Model Controller** to communicate with iDlaLayer and then data information is sent to **Data Controller** component. (2) *Data Classification* maintains the information of this dataset by setting and updating its life stage, and *Updating Controller* decides whether to make model updating. (3) Once the updating decision is made, iDlaLayer informs the backend framework–TensorflowOnSpark to perform model training. (4) Meanwhile, *Data Selection* is ready to send combined data samples to TensorflowOnSpark. (5) After the backend framework finishing retraining model with new data samples, it notifies **Model Controller**'s *Training Profiler* module to update the training time profile for that application. Then *Updating Controller* evaluates contributions the data

samples have made as well as updates data's life stage. The updated model can finally be shipped to model serving systems for further inference requests.

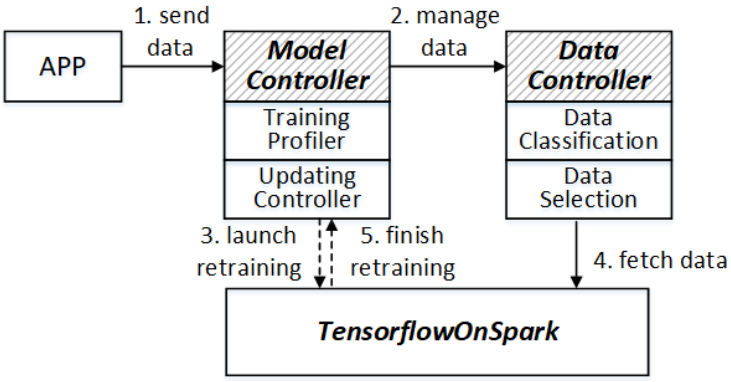


Figure 5.4: The training workflow of iDlaLayer.

iDlaLayer uses a process *allocate_mem* to initialize memory space for maintaining data. A function named *serve_app* is launched in **Model Controller** component to handle training data samples for each application. When new data arrives, this process transfers data to **Data Controller** component by calling *reciv_data* function running in it. *devi_data* is responsible to decide and update data's life stages based on its contribution to model updating. Based on the proposed *Fuzzy Model*, we experimentally determine the data life stage ranges regarding to the distribution of each data's contribution to the model quality. We implement DLA in *Updating Controller* module determining when to perform model updating. It follows the process we describe in Algorithm 1. Once an updating decision is made by *launch_update* in *Updating Controller*, building training data sample procedure starts to work by invoking *build_sample*. It is a lightweight process with quick response and prompt processing features to guarantee a solid service to backend systems. Simultaneously, the backend also gets updating notification and begins to fetch data sample from **Data Controller** component. After backend finishing model updating with new data, it recalls *log_profile* function in *Training Profiler* to update the training time cost by application and the contributions new data has made. Meanwhile, *devi_data* is informed by *log_profile* to update data life stages it maintains.

Table 5.1: Experiment Applications

Application	Algorithm	Dataset	Input
Pattern Recognition	CNN	MNIST	11.6MB
Classification Prediction	DNN	Criteo	1GB
Recommendation System	PageRank	wikipedia	243 MB

5.5 Evaluation

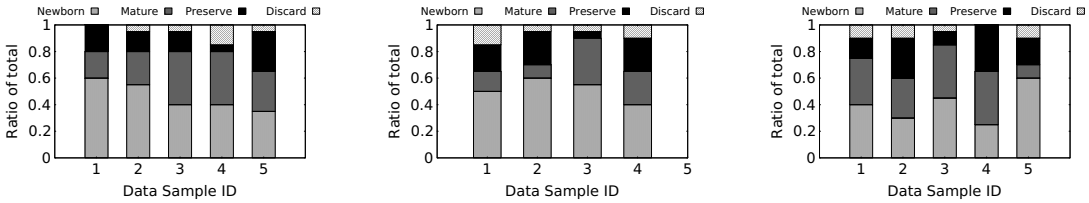
5.5.1 Experiment Setup

The evaluation testbed consists of 15 nodes, each of which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR3 RAM, running Ubuntu 16.04 LTS operating system with kernel version 4.0, Scala 2.10.0, and Hadoop YARN 2.8.0 for cluster management. One node serves as the *master*, and all the other 14 nodes serve as *slaves*. These nodes are connected with Gigabit Ethernet.

We select three representative applications: the pattern recognition with MNIST dataset [48], the classification prediction based on Criteo [49] and the recommendation system with PageRank. Table 5.1 shows the detailed attributes of the applications we used in experiments. In our implementation-based experiments, we set up the scenario in which data are continuously generated and fed into the system. Each sample data has a timestamp in the trace so that they roughly follow the behavior of real-world data. The performance evaluation mainly focuses on (i) training cost, (ii) ratio of data life stages, (iii) model quality, and (iv) overhead.

5.5.2 Effectiveness on Ratio of Data Life Cycle

Figure 5.5 depicts data ratios of different life stages on five randomly time points from the three applications, i.e., MNIST, Criteo and PageRank. We use stacked histogram to denote the ratio of data life stages. Four patterned columns represent the four data life stages we defined above. At each time point, it is obvious that ratios of data life stages vary a lot since the solution to our objective function on finding better data combination ratios changes over time. There are a few conditions that a data life stage does not show up in



(a) MNIST (b) Criteo (c) PageRank

Figure 5.5: Ratio of data life stages on randomly chosen time points from MNIST, Criteo and PageRank.

the combinations e.g., the time point 1 of MNIST (Figure 5.5(a)) and the time point 4 in PageRank (Figure 5.5(c)). These cases illustrate the data in *Discard* stage makes no contributions to improving model performance so that they are not included in combined data samples. Figure 5.5 also displays that these combinations of data samples keep changing at runtime and each application shows unique behavior in terms of combined data samples. An effective ratio at a time point may not be valuable for the others. iDlaLayer is capable of building training data samples with data from different life stages in terms of how much contribution they could make to model updating.

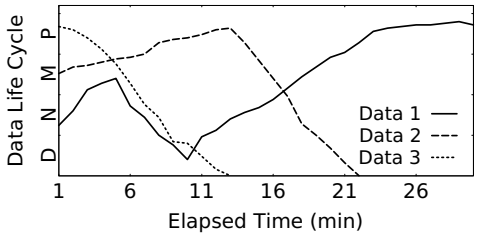
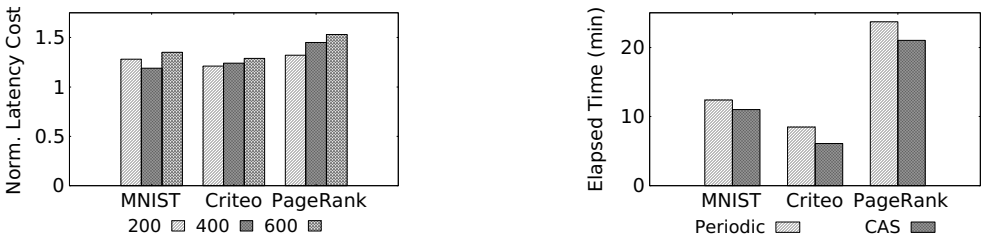


Figure 5.6: Dynamic data contributions over time.

We further monitor dynamic data life cycle changes at runtime. Figure 5.6 depicts traces of three data samples from PageRank. y-axis represents four ranges illustrated by data life stages (D=Discard, N=Newborn, M=Mature and P=Preserve) while x-axis is a randomly selected time window. In the experiments, we use life stages to describe contribution changes of data sample instead of how much contributions they have made to model quality. Experimental results show each data sample traverses four different stages during the whole life cycle. Moreover, there are tendency variances among data samples over



(a) Incorporation latency (b) Incorporation elapsed time

Figure 5.7: (a) displays data incorporation latency of DLA normalized by Optimal. (b) shows elapsed time from Periodic and iDlaLayer.

time, which is displayed by curves fluctuations. Even at the same time point, life stages of different data samples vary dramatically, such as at time point 12, *Data 1* just arrives and is allocated in Newborn stage, *Data 2* is at the end of its life cycle and *Data 3* contributes a lot and is in the Mature stage. Based on these observations, DLA combines training data samples in a real-time and dynamic manner based on life stages of data, which guarantees the overall data contribution to model quality is high and maintains more valuable data for future model updating while getting rid of dirty data.

5.5.3 Effectiveness on Training Cost Reduction

To illustrate performance differences in training cost, we compare our proposed solution with Optimal strategy. We conduct real trace-based simulations to demonstrate the performance difference in latency. Given the fact that the optimal training follows a linear pattern to the size of data samples, we run this experiment on 200, 400 and 600 data samples randomly selected from the datasets. Figure 5.7(a) depicts the data incorporation latency cost of our method, which is normalized by Optimal offline training strategy. By checking the results from those three applications, we observe that DLA also follows a linear relationship to the size of data sample. Moreover, our method performs very closely to Optimal offline training strategy in latency (which is about 1.32x on average).

We then compare Periodic updating and iDlaLayer deployed on a psychical cluster with 15 nodes. Figure 5.7(b) plots the performance (elapsed time in incorporating data) of Pe-

riodic and iDlaLayer. The improvements incurred by iDlaLayer are 11.3%, 28.2% and 15.2% for MNIST, Criteo and PageRank respectively. Criteo benefits more than the other two applications. The reason is iDlaLayer could maintain data based on its life stages with more useful data combined into data samples and Criteo demands more duplicated data in model training. The figure shows that PageRank requires more time in training model while our method still decreases its total cost.

5.5.4 Effectiveness on Model Quality

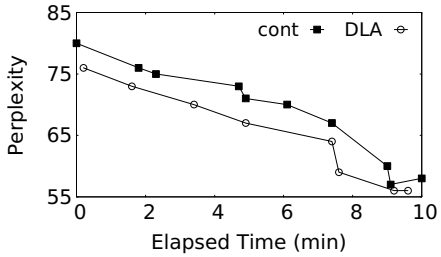
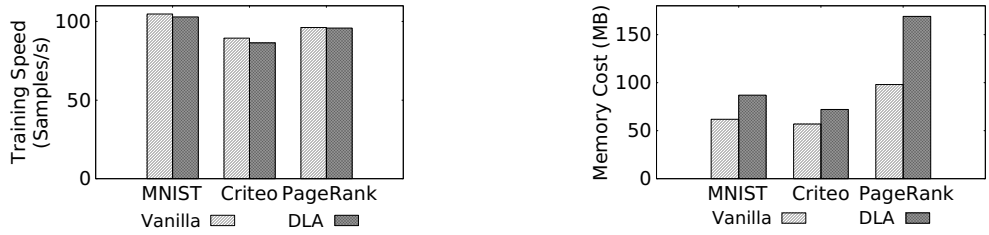


Figure 5.8: Quality of MNIST.

Model quality ultimately decides how well a prediction could be achieved. Specifically, we choose MNIST application and compare its quality between continuous (*cont*) and iDlaLayer architectures. Model quality is measured by *perplexity* as we discussed in Section 2.2. Figure 5.8 presents the result of a 10-minutes time window, where each dot represents the end of one model updating procedure. Overall, the perplexity of *cont* and iDlaLayer drops with the incorporation of new data. iDlaLayer's perplexity drops more than *cont* as it uses more beneficial data samples in model updating. Besides, it has fewer data points in the dotline comparing with *cont* because some retraining instances are abandoned and merged into others. Consequently, every model updating procedure in iDlaLayer covers more data and brings a better quality, which is about 5% improvement, as shown in Figure 5.8.



(a) Training speed (b) Memory consumption

Figure 5.9: The training speed and the memory consumption of Vanilla TensorflowOnSpark and iDlaLayer.

5.5.5 Overhead

To analyze the overhead of iDlaLayer, we use the vanilla TensorflowOnSpark as a baseline and conduct experiments to measure (i) training speed, which measures how many samples the system could process per second and (ii) memory consumption, which scales how much memory resource it takes to realize the retraining strategy. Figure 5.9(a) depicts the training speed comparison between the vanilla TensorflowOnSpark and iDlaLayer. iDlaLayer results in 1.8%, 3.5% and 0.7% slowdown on MNIST, Criteo and PageRank respectively. We can explain those slowdowns with DLA's strategies on combined data samples and model updating decisions, which take a short time before the training procedure begins. Overall, slowdown in training speed, which is 2% on average, does not significantly affect applications' performances. Instead, iDlaLayer decreases more time spent in finishing the whole applications. Figure 5.9(b) shows memory consumption of the vanilla TensorflowOnSpark and iDlaLayer. For all three applications, iDlaLayer consumes 28.7%, 20.8% and 42.1% more memory resource in running MNIST, Criteo and PageRank respectively. The average memory overhead introduced is about 30.5% (23 MB), which is negligible to the whole system-level memory resource. Considering the performance improvement achieved by DLA, such memory overhead is an acceptable trade-off between space and time consumption.

CHAPTER 6: REFERENCE DISTANCE AND LOCATION BASED DATA MANAGEMENT ON HYBRID MEMORY SYSTEM FOR DEEP LEANING

6.1 ReDL Architecture Overview

In this section, we propose a runtime system that implements ReDL, a thin middleware layer between applications and backend deep learning frameworks.

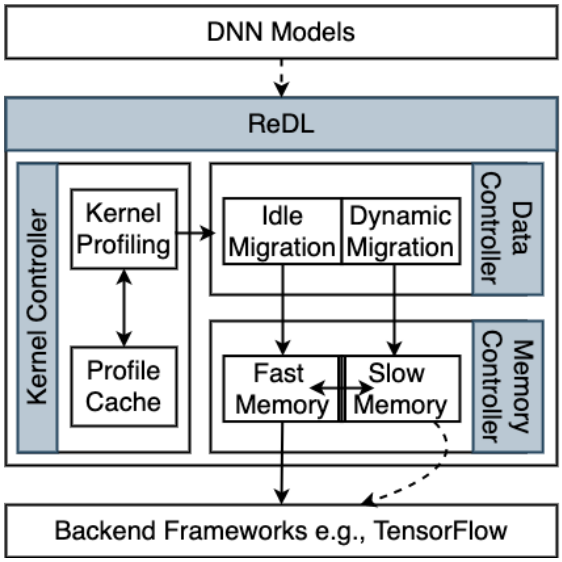


Figure 6.1: Architecture Overview.

An overview of the proposed architecture is shown in Figure 6.1. The key contribution relies on a novel reference distance and location based data management strategy (ReDL) on hybrid memory system for DNN applications. RedL can place data objects (tensors) into fast or slow memory and migrating data objects among them based on the liveness of data objects. More details on data management strategy would be discussed below. As Figure 6.1 shows, our proposed architecture mainly consists of three components: *Kernel Controller*, *Data Objective Controller*, and *Memory Controller*.

Kernel Controller has two functional modules. The Kernel Profiling collects memory

access information of each kernel, records the related data objects (tensors), calculate the kernel's execution time in each layer, and decides the liveness of the kernel. The profiling process only needs one training step to obtain the information and then update the profile information in Profile Cache. The Profile Cache is a software cache reserved to record profiled kernels. Since profiling of some DNN models may take longer time in training and DNNs computation graph usually includes a lot of identical kernels, Profile Cache is necessary. Besides, by reserving a specific profiling cache, the profiling step for any DNN model only needs to be conducted one time.

Data Objective Controller is driven by the profiling information and implements the ReDL method we proposed. Short-lived data objects are processed by *Idle Migration* while long-lived ones are managed by *Dynamic Migration*, respectively. The short-lived data object is placed into the fast memory with a contiguous free space by Idle Migration, and some of them will be freed when the space is in a tension status. This method decreases some unnecessary data movement caused by the short liveness of data objects. We adopt a Dynamic migration strategy for long-lived data objects, which is to migrate data among the fast and slow memory periodically. In a migration period, the Dynamic Migration module prepares the requested data objects for the next coming period based on the static DNN computation graph. To handle the case that requested data objects are not timely migrated in the fast memory, we propose **Direct Slow Memory Access (DSMA)**, which breaks the barrier between computing units and slow memory by allowing computing units to directly access data objects in the slow memory.

Memory Controller consists of two modules. The *Fast Memory* manages data placed in fast memory while the *Slow Memory* maintains data objects in slow memory space. Besides, *Fast Memory* and *Slow Memory* communicate with each other to finish data migration between them. Data Objective Controller sends out all the data management information with a periodical heartbeat connection protocol. ReDL controls data migrations among fast and slow memories overlap DNN application execution, such that the application perfor-

mance is not affected.

6.2 Kernel Profiling

Kernel Profiling inspects the DNN computation graph to extract the types and computation orders of kernels in the computation graph, the producer(s) and consumer(s) of each kernel, and the data objects (tensors) it involves. It collects the profiling information of the kernel and tensor, such as its liveness, accessing time, and execution time by changing its inputs and outputs in fast and slow memories. Since the chosen location of input and output tensor location is tentative, it might take a couple of training steps to decide the best execution time of each kernel. We use the *Profiling Cache* to record the profiled kernels so that the profiling step for any DNN graphs only needs to be conducted once.

We need to minimize the overall execution time of DNN computation graph with limited memory size. In the static DNN computation graph, computation kernels are executed sequentially. Therefore, there is not data object movement during the training step. We formulate the objective function as:

$$min \sum_{k \in \kappa, t \in \tau} \rho_{k,t}$$

$$\text{s.t.} \begin{cases} \sum_{t \in \tau} f_{(t,k)} + \sum_{t \in \tau} s_{(t,k)} \geq \tau & (1) \\ \sum_{t \in \tau} f_{(t,k)} \leq FS & (2) \\ \sum_{t \in \tau} s_{(t,k)} \leq SS & (3) \end{cases}$$

where *k* is a kernel in the set of all kernels in a computation graph and ρ_k is the expected execution of kernel *k*. τ is the collection of all tensors for a kernel *k* and *t* is a tensor in τ. ρ_k highly depends on the selection of locations to host input and output tensors for kernel *k*. The location of the related inputs and outputs is important due to the dependencies among kernels. As shown in Figure 2.3(b), the optimal case is inputs are placed in the slow memory and outputs in the fast memory. The main constraints are on the amount of memory used by the computation graph. Constraints (1) denotes the total memory usage in

the fast and slow memory is no smaller than the total size of tensors for a kernel. Constraints (2) and (3) represent the memory space used in the fast and slow memory, respectively.

To solve the objective function, we use the linear and integer programming theory [99], which uses the profiling information to automatically optimize the placed locations of data objects with memory size as the constraints.

6.3 Idle Migration

Idle Migration manages short-lived data hosted in the fast memory aiming to guarantee a smooth training process by decreasing unnecessary suspense for data fetching. During the DNN model training steps, the short-lived data is not accessed too frequently when compared with the long-lived data object. However, Figure 2.9 shows a huge number of short-lived data objects across the full DNN training step, which indeed has a non-negligible impact on the training performance.

6.3.1 Managing Data Objects

A continuous space is allocated in the fast memory to host data objects with short liveness. This memory space is reserved for data with shorter liveness and some newly migrated long-lived data objects. All the short-lived data objects in the fast memory are not considered for migration when a memory tension occurs because their quantity is large. The migration of short-lived data in other memory can take a lot of time and detrimental to the training process. However, long-lived data objects are supposed to be migrated to the slow memory if there is no enough space in the fast memory because these data objects have a higher probability of being accessed again within the training steps. The continuous fast memory space is assigned by the memory controller at the very beginning of every migration period to host data objects for this current training iteration. With this operation, ReDL ensures that there is enough space to hose short-live data objects. The access information of data objects is also collected during the migration period to achieve efficient memory utilization and better application performance.

When a new short-lived data object is required, ReDL first checks the free size of the fast memory. If there is enough size to hold the data objects, it is directly placed. If not, some short-lived data objects are selected as the victims to be freed, and then the migration of new data objects is processed. The selection of victims is based on the liveness. Short-lived data objects with shorter liveness are chosen. By doing this, the side-effects on application performance is lighten. At the end of the current migration period, ReDL updates the metric information such as liveness, accessing times for data objects and free space size, memory utilization, and data locality for the fast memory used in the next migration period.

Algorithm 4 Idle Migration

Input: $profiling\_info$, $fast\_mem\_size$, $migration\_seq$
Output: $data\_info$, mem_info

```
1:  function AllocFastMem(fast_mem_size)
2:      if free_size > fast_mem_size
3:          mem_pts = fastMalloc(fast_mem_size)
4:          free_size = free_size - fast_mem_size
5:      return mem_pts
6:  end function
7:  function ParseProf(profiling_info)
8:      data_info = parseInfo(profiling_info)
9:      return data_info
10: end function
11: function FastDOCtrl(mem_pts, data_info)
12:     //New Data Paged in
13:     if mem_pts is empty or enough space
14:         mem_pts = placeData(data_info)
15:     else freeVictimData()
16:     return mem_pts, data_info
17: end function
18: function UpdateInfo(profiling_info)
19:     profiling_info = updateInfo(mem_pts,data_info)
20:     return profiling_info
21: end function
```

6.3.2 Data Movement Implementation

Algorithm 4 depicts the main procedure of *Idle Migration*, mainly including *AllocFastMem*, *ParseProf*, *FastDOCtrl*, and *UpdateInfo* functions. (1) The fast memory is allocated

at the beginning of each data migration period. Function *AllocFastMem* (Line 1-6) finishes smoothly applying for a continuous space from the fast memory if the available space satisfies the requested memory size and updates the size of free fast memory. (2) Then *ParseProf* function parses the profiles of data objects generated by profiling modules. In this step, the basic accessing information of each data object (tensor) is abstracted, such as liveness, execution time, input (producer), and output (consumer). According to the parsed information, the short-lived data objects are preferentially placed in the just allocated fast memory space. *ParseProf* (Line 7-10) also handles the migrations of the long-lived data objects from the slow memory. (3) When a new data object request arrives and is not in the fast memory, data migration is triggered. *FastDOCtrl* (Line 11-17) first checks whether the fast memory space is empty or there is enough space for the new data object. If the checking result is true, this data object is directly placed in fast memory. Otherwise, *FastDOCtrl* traverses all the live data objects in the fast memory to find victims that are supposed to be freed. Victims are the data objects with much shorter liveness and on longer accessed in the current migration period. (4) The *UpdateInfo* (Line 18-21) function updates the metric information of the fast memory size and data objects. And this information helps make data management decisions in the next migration period.

The idle migration policy above tackles the issue of migrating short-lived data objects back to the slow memory space if they are no longer requested. Idle Migration decreases unnecessary data migrations, which result in performance loss and waste memory bandwidth. It is a huge waste of the fast memory space if short-lived data objects keep occupying the limited valuable fast memory space. Furthermore, deciding the migrations of short-lived data objects is time-consuming but cannot always guarantee accuracy since collecting memory accessing information takes time, and calculating the total number of memory access for data objects may be wrong. Idle Migration overcomes the limitations with the DNN domain knowledge and all the essential information for making migrations has been obtained in the profiling process.

6.4 Dynamic Migration

Dynamic Migration controls migrations of long-lived data objects in the slow memory. It uses the scalable migration period to decide the amount of long-lived data objects placed in the slow memory and the frequency of data movements between fast and slow memories. Besides, Dynamic Migration supports directly accessing the slow memory for the case that requested data is not timely migrated in the fast memory. The critical operation in Dynamic Migration is determining an optimal migration period size that brings the best DNN training performance. In ReDL, a training step is equally divided into many migration periods to guarantee a flexible control on long-lived data objects migrations. We use the layers in the static DNN computation graph as the metrology to define the migration period. The layer-based migration period usually ensures the completion of kernel operation at the end of each period because no operations are running across layers. The static DNN graph structure is fixed at the compiling step, which provides the probability to find the optimal layer-based migration period. Besides, every layer has its own associated computation sequence which would display a unique memory access behavior. Our proposed layer-based migration period leverages the memory access pattern obtained in the profiling step to lead data migrations.

6.4.1 Determining Migration Periods

Figure 6.2 depicts a general example of a layer-based data migration period. In this example, two randomly consecutive migration periods, t_1, t_2, are displayed.

Data objects migration for period t_2 is triggered at the beginning of the t_1, aiming to migrate most requested shored-lived data objects to the fast memory and long-lived ones in the slow memory before the second period starts. This operation occurs during the whole period so that data migration overlaps with DNN training and the overhead of data migration is further neutralized. Given the fact that the size of fast memory space is limited, memory tension is inevitable, which is shown as "Trigger migrations when memory tension

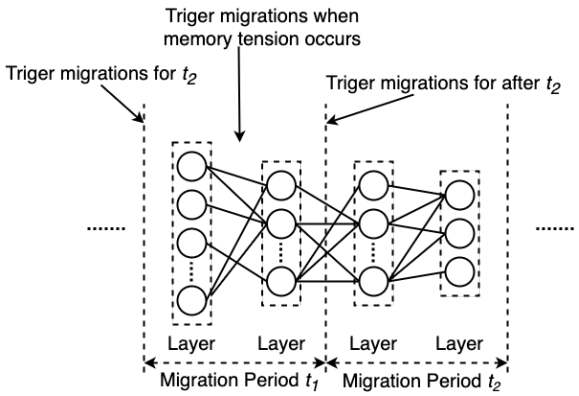


Figure 6.2: Layer-based data migration period.

occurs" in Figure 6.2. When this case happens, some unused short-lived data objects are first freed, and then the long-lived ones that are not accessed in the current period are migrated back to the slow memory. Such a strategy is applied to save the fast memory space as much as possible. At the end of migration period t_1, another data object migration is triggered for the period after t_2. A similar operation is conducted until the training phase is finished.

Determining an optimal migration period is a dilemma. If the migration period is too large, the amount of data objects to migrate can exceed the free memory space, especially for the fast memory. If we adopt a small migration period, then the possible execution time to overlap with DNN training is shorten. So the migration period should not be too short; otherwise, the migrations of data objects to the right memory space cannot be timely finished before the next migration period begins. A trade-off is to make between large and small migration periods.

To tackle this problem, we formulate the objective function as follows:

$$min \sum_{l \in \gamma} L_l(I)$$

where γ is the collection of all layers *l* from DNN computation graph, L_l is the expected execution time of layer *l*, and *I* is the optimal migration period for the DNN. The target is to *minimize* the total execution time of the whole computation graph with the limited memory space. Note that L_l depends on the memory size and data objects migration period.

For L_l, we can express it as:

$$L_l(I) = p_l(I) + m_l(I) \qquad (6.4.1.1)$$

where $p_l(I)$ is the computation time and $m_l(I)$ is data objects migration time. To find the optimal migration period *I*, we find that $p_l(I)$ and $m_l(I)$ have positive correlations with the migration period *I*. The large the migration period is, the longer time takes by $p_l(I)$ and $m_l(I)$. Equation (6.4.1.1) is subject to the following constraints:

$$\begin{cases} p_l(I) \leqslant S - F(I) & (1) \\ m_l(I) \geqslant p_l(I)/BW & (2) \end{cases}$$

where *S* is the total fast memory size, *F* is the total fast memory space reserved by the short-lived data objects, and *BW* is the migration speed determined by bandwidth between the fast and slow memories. *F* denotes the function of a migration period *I*. In our method, we assume that each migration period has its specific *F*. According to the profiling results, *F* is relatively stable. *S* is a constant, so $S - F(I)$ is close to constant. $p_l(I)$ and $m_l(I)$ are also monotonically increasing functions of *I*. Hence, constraints (1) and (2) build the upper and lower limitations in determining the migration periods.

Although Equation (6.4.1.1) and its constraints reveal the inherent trade-off among large and small migration periods, it still needs a dedicated algorithm to find the optimal migration period. In ReDL, we adopt an iterated greedy algorithm [100] [101] to determine the optimal period at runtime. When the profiling step is finished, we start with the migration period of the median of the total layers and then test if this period satisfies the constraints. If the test result is positive, the optimal migration period is determined. Otherwise, we will first repeat this process with a new migration period by adding 1 layer to the current period size. And then test the case by deducting 1 layer from the median number of total layers. During this optimizing round, three training iterations are included. In the next round, the algorithm tests the migration periods by adding or deducting 2 to the median number. A similar round is repeated until the optimal migration period is found. We measure the per-

formance from different migration periods and select the one with the best performance in the following training procedures. We must ensure that data placement within any optimizing round is the same in order to obtain accurate comparison results. The same data placement is easily guaranteed due to the repetitive and predictive execution behavior in DNN model training. It might take a couple of rounds to determine the optimal migration period and result in some performance loss, but the total overhead is not large because this case does not often happen and performance loss is compensated in the remaining training steps.

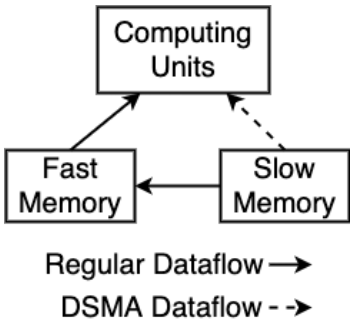


Figure 6.3: Direct Slow Memory Access (DSMA).

6.4.2 Direct Slow Memory Access

Another case to consider is the optimal migration period fails to fit some of the computation phases, because there is no enough time for migrating requested data objects from the slow memory to the fast memory before the upcoming period starts. To tackle this problem, we propose **Direct Slow Memory Access (DSMA)**, which breaks the barrier between computing units and slow memory by allowing computing units to directly access data objects in the slow memory. Figure 6.3 shows a simple dataflow of DSMA. In the regular dataflow, data objects in the fast memory are directly accessed by computing units such as CPU, GPU, and FPGA to execute DNN model training. Data objects saved in the slow memory are first migrated in the fast memory and then computing units can use them. With DSMA, computing units can directly access data objects in the slow memory without first migration in the fast memory. Application performance from DSMA cannot compete

with the regular access mode, but DSMA does not occur frequently and its adverse impacts are negligible.

Algorithm 5 Dynamic Migration

Input: $profiling\_info$, $fast\_mem\_size$, $migration\_seq$
Output: $data\_info$, mem_info

```
1:  function AllocSlowMem(fast_mem_size)
2:      if free_size > fast_mem_size
3:          mem_pts = fastMalloc(fast_mem_size)
4:          free_size = free_size - fast_mem_size
5:      return mem_pts
6:  end function
7:  function ParseProf(profiling_info)
8:      data_info = parseInfo(profiling_info)
9:      return data_info
10: end function
11: function FindOptPerid(data_info)
12:     opt_period = findPerid(data_info)
13:     return opt_period
14: end function
15: function SlowDOCtrl(opt_period, mem_pts, data_info)
16:     if mem_pts is empty or enough space
17:         mem_pts = placeData(data_info)
18:     else freeVictimData()
19:     return mem_pts, data_info
20:     //Data Migrated to fast memory
21:     if opt_period is valid
22:         mem_pts = migrateData(data_info)
23:     else accessSlowMem(mem_pts)
24: end function
25: function UpdateInfo(profiling_info)
26:     profiling_info = updateInfo(mem_pts,data_info)
27:     return profiling_info
28: end function
```

Algorithm 5 depicts the main modules of *Dynamic Migration*. The functionalities *AllocSlowMem*, *ParseProf* (Line 7-10) and *UpdateInfo* (Line 25-28) are similar to *Idle Migration*. *SlowDOCtrl* (Line 15-24) manages the long-lived data objects. The newly coming long-lived data is placed in slow memory. The migration operation to the fast memory happens only when it is during a valid migration period and there is enough free space in

the fast memory. We implement our proposed iterated greedy algorithm in *FindOptPerid* (Line 11-14), aiming to find the optimal migration period. As discussed in the above context, determining the optimal migration period might take a couple of rounds and cause some performance loss. We implement Direct Slow Memory Access (DSMA) in *accessSlowMem* function (Line 23) in *SlowDOCtrl* to handle the case that data migration to the fast memory cannot be finished due to the migration period does not fit to some layers. With DSMA, the training step still continues by directly accessing long-lived data objects in slow memory and does not need to migrate date from the fast memory. Overall, Algorithm 5 does not bring large overhead because these special cases do not happen often. Hence a huge number of training steps or testing rounds to determine the optimal migration period is not too needed.

In ReDL, since the long-lived data objects are placed in the slow memory and their total size is usually much larger than the short-lived ones, the size of this memory is allocated tens of GB to host data objects for the whole DNN model training steps.

6.5 Implementation

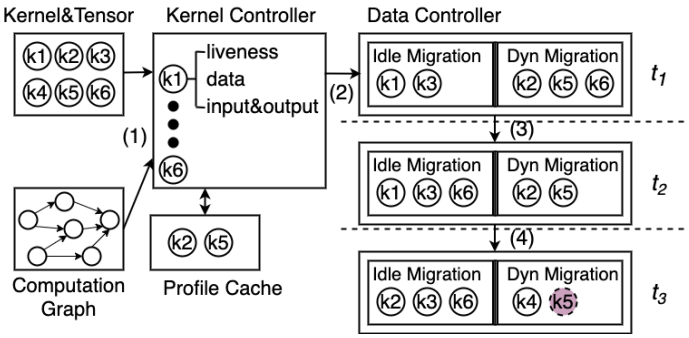


Figure 6.4: An Example of Workflow with ReDL

Figure 6.4 illustrates the detailed workflow of our work. In this example, we include three data migration periods: t_1, t_2 and, t_3. The profiling process is performed before t_1 begins. Step (1) Data objects (in this paper, data object and tensor represent the same item), six kernels in this figure {*k1*, *k2*, *k3*, ..., *k6*}, for training DNN model and the corresponding static computation graph are fed into *Kernel Controller*, which profiles the basic metric

information of each kernel such as its execution time, liveness, input (producer) and output (consumer), and access records of its related data objects. Meanwhile, *Kernel Controller* searches *Profile Cache* to check if any kernel's profiling information has been profiled in the prior training iteration. If the search returns a positive result, the existed profiling information is used; otherwise, a new profiling record is written into this cache. Step (2) Then the first data migration period t_1 starts with *Data Objective Controller*. As Figure 6.4 shows, *k1*, *k3* are short-lived data objects and allocated in the fast memory space by *Idle Migration* and long-lived kernels *k2*, *k5*, *k6* are placed in the slow memory with *Dynamic Migration*. As the current training iteration executes, the data object requirements for the next iteration might change. Step (3) The application enters data migration period t_2, which is overlapping with the application execution. *k6* is migrated from the slow memory to the fast memory. This migration should be made timely before *k6* is needed by the next training iteration. In t_2, the data migration mainly occurs between different memory hardware, and no new data objects are placed in the fast or slow memory. Step (4) In t_3, *k4* is newly migrated in the slow memory based on data requirements analyzed from the computation graph. *k1* is freed because it finishes all its lifetime and no other kernels need it. A special case is that the required data object *k5* is not timely migrated in the fast memory. In this particular case, the application continues executing by directly accessing *k5* from the slow memory. Accessing the slow memory is relatively slower than the fast memory, but it is still a memory-access operation and faster than migrating the needed data objects from the local disk.

6.6 Evaluation

6.6.1 Experiment Setup

We study the performance of ReDL on a single machine, which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR4 RAM. Table 6.1 shows the testbed environment in our experiments. Persistent Memory Block Driver (PMBD) [53] simulates a slow memory while DRAM serves as the fast memory so that a hybrid memory system is built.

Table 6.1: Overview of Experimental Environment

CPU	Intel Xeon(R) CPU E5-2630v4
DRAM	64 GB DDR4
Fast Memory	Bandwidth: 36 GB/s Latency: 90ns
Slow Memory	Bandwidth: 18 GB/s Latency: 200ns

Table 6.2: Summary of selective benchmarks.

Benchmark	Dataset	Batch size	Model size
ResNet50 V2	CIFAR-10	128	98 MB
LSTM	PTB	20	106 MB
VGG 19	CIFAR-10	64	549 MB
Inception V3	MNIST	64	92 MB

We compare the performance of ReDL with NUMA [54] and OS-Integrated Multi-level memory management system(OIM) [21]. In NUMA [54], when new memory is allocated, it will occur in the fast memory if free space is available; otherwise, it will only occur in the slower memory node. OIM improves page migration performance by launching 4 threads for paralleling page copy and 8 threads for concurrently conducting page migration, and it also optimizes the location of page every five seconds [21]. Unless specified otherwise, all the experiments are conducted on a pure CPU platform and the total size of the fast memory is configured to be 20% of the peak memory consumption of each DNN model. We will leave the CPU and GPU computation in the future work.

We select four popular DNN workloads to benchmark our work. Table 6.2 illustrates some selective workloads and their parameters. TensorFlow [52] is used to implement ReDL and test its performance with these four workloads: ResNet50 V2 [102], LSTM [56], VGG 19 [21], and Inception V3 [103]. Each workload is run until its execution time in a migration period is stable. Given that these workloads do not have data dependent relations, their performance will be relatively stable with the first couple of computation iterations finished. For Tensorflow, we set its inter-op parallelism and intra-op parallelism to be 20 so that this is in consist to the physical cores in our testbed, so that all the experiments are conducted on the CPU with one thread per physical core. The performances of our work

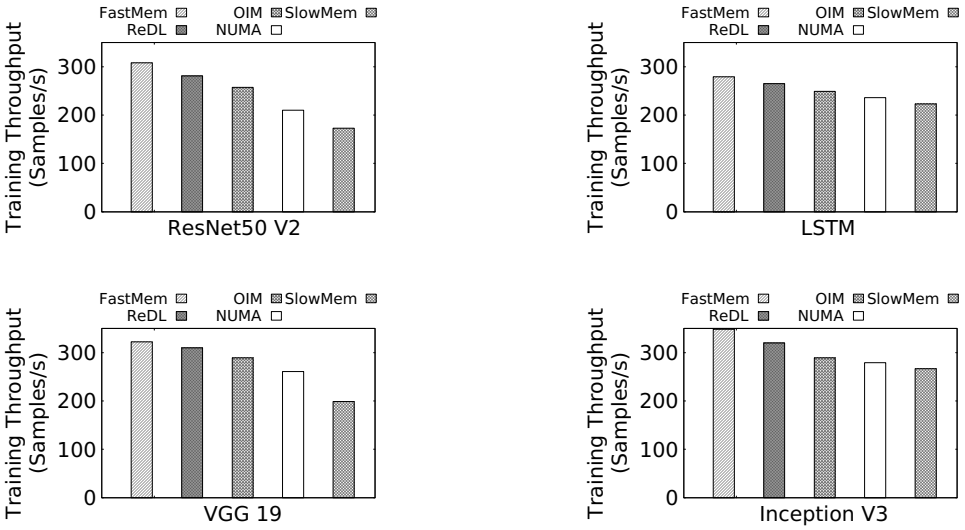


Figure 6.5: Training throughputs from FastMem, ReDL, OIM, NUMA and SlowMem

are mainly evaluated in (i) training throughput, (ii) training speedup, (iii) data locality, and (iv) overhead and scalability.

6.6.2 Training Throughput

Figure 6.5 depicts the performance of DNN model training throughputs. We compare ReDL with the following cases: FastMem (fast memory only system), OIM, NUMA and SlowMem (slow memory only system). The size of fast memory is configured to be 20% of each workload's peak memory consumption. The figures show that the performance differences between ReDL and all the fast memory cases are not very obvious. The maximum difference is 9.6% from ResNet50 V2 while the minimum is 3.9% from VGG 19. Overall, ReDL has an average 6.9% performance improvement. ReDL averagely outperforms OIM with 8.4% (up to 11% in Inception V3) and better than NUMA by 20% on average (up to 34% in ResNet50 V2). The improvements are bought with the fact that ReDL optimizes data migrations between fast and slow memory in tensors and conducts migrations only within migration periods. ReDL also has an average 20% increase compared with NUMA.

We further compare the number of data migrations in ReDL, OIM and NUMA. Table 6.3 shows the number of page migration with four workloads. We use the page as the unit to

Table 6.3: Migrated pages in one training iteration.

	ResNet	LSTM	VGG	Inception
ReDL	2391343	399635	6132049	354013
OIM	2173450	401527	5904241	352702
NUMA	1971243	312307	3973926	250971

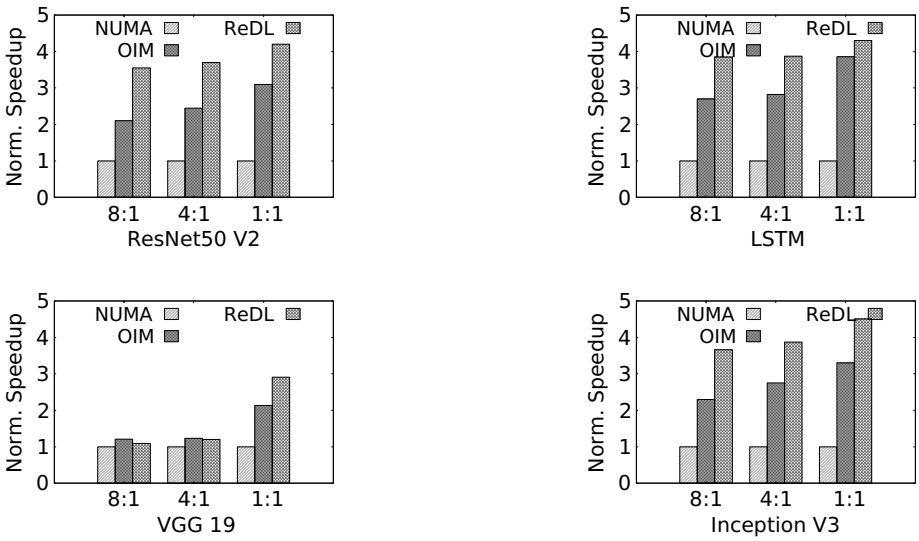


Figure 6.6: Training speedup normalized to NUMA under different ratios between fast and slow memory.

count migration because OIM and NUMA use it to conduct data migrations. ReDL has an average of 3.5% and 36% more migrations compared with OIM and NUMA, respectively. Frequently data migrations allow ReDL to make the best use of fast memory so that better application performance is achieved. Besides, those migrations are overlapped with DNN training steps which further increase the performance

6.6.3 Training Speedup

Figure 6.6 shows the speedup achieved by ReDL normalized to training with NUMA. The x-axis denotes the ratio of slow to fast memory used to train the four workloads. In this experiment, we apply 40GB from the system memory and use PMBD [53] to simulate it as the fast and slow memories following different ratios above. We notice that NUMA's performance is poor compared with OIM and ReDL in all ratios: 8:1, 4:1, and 1:1. With

more memory space is allocated as fast memory, performance improvements by NUMA are not so dramatic. This is because NUMA first allocates data objects in the fast memory without considering the computation sequence until the fast memory is full. Besides, in NUMA, the long-live data objects are placed in the fast memory resulting in more data migrations. OIM tackles this issue via whole page migration and parallel migration strategy, but its performance improvement is not as good as ReDL. ReDL has better performance under any memory ratios because it is aware of the liveness of data objects and based on them to optimize data migration between fast and slow memory.

In Figure 6.6, VGG 19 is an exception because it have an extremely large second convolution layer, which demands more memory space to host its related data objects. When the fast memory is small (i.e., in the cases with ratios 8:1 and 4:1), some of the data objects must be placed in the slow memory, incurring a performance loss. When there is enough fast memory, we note a huge performance improvement, as it is illustrated in the figure, the performance jumps high from the 4:1 ratio to the 1:1 ratio.

6.6.4 Data Locality

Figure 6.7 illustrates the data localities from NUMA, OIM and ReDL. In this experiment, we define data locality as the percentage of requested data objects in the fast memory. Higher data locality implies fewer data migrations and, in turn, reflects the efficiency of data management strategy. In Figure 6.7, we monitor the workloads' activities until their performance is stable, which is reflected by a relatively smooth line. We can observe that the stable time for each workload differs due to the differences in their computation graphs. Furthermore, a couple of peaks and valleys in each line, which display locality fluctuations. The valleys between any two consecutive peaks illustrate data migration. Note that our proposed work, ReDL is always above NUMA and OIM with an average of 19% and 11% improvements, respectively. ReDL also delays the occurrence times and frequency of valleys because it can timely migrate data objects in the fast memory. There is an outlier in the experimental results of VGG with NUMA; a spike occurs around the time points

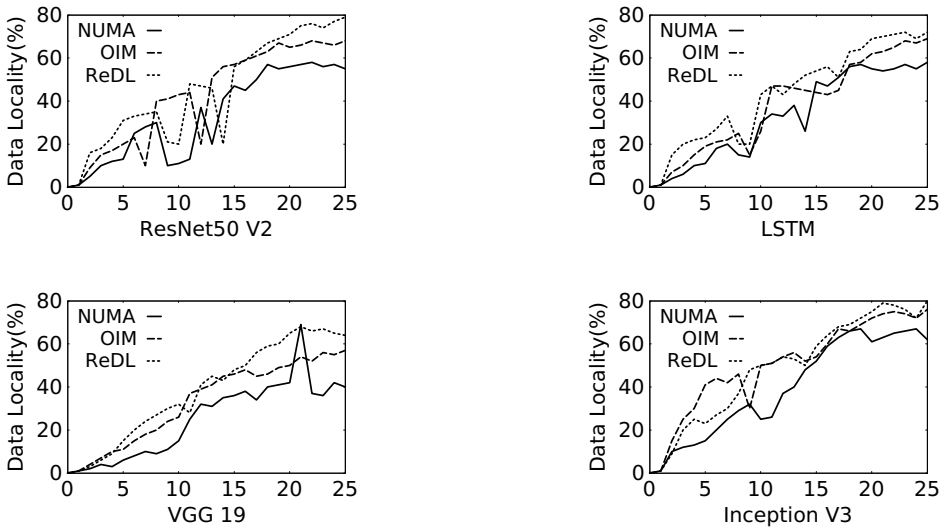


Figure 6.7: Data localities from NUMA, OIM and ReDL.

20. We can explain this exception with its special large convolution layer, which demands more memory while the memory management policy in NUMA is not so efficient.

6.6.5 Overhead and Scalability

To analyze the overhead of ReDL, we use NUMA as the baseline and conduct experiments to measure peak memory consumption in NUMA, OIM and ReDL, which scales how much memory resource it takes to conduct DNN model training. Figure 6.8(a) shows the memory cost of four workloads running on in NUMA, OIM and ReDL. Overall, ReDL consumes 6% and 3% more memory compared with NUMA, OIM. This is because ReDL needs some extra memory to conduct profiling and optimizing steps. For workload Inception V3, its peak memory consumption is relatively stable in all platforms. Because the model size of Inception is small and its computation graph is simple so that the memory requirement is not large. The average memory overhead introduced is about 5% (about 3GB), which is negligible to the whole system-level memory (64GB in our testbed). Considering the performance improvement achieved by ReDL, such memory overhead is an acceptable trade-off between space and time consumption.

Figure 6.8(b) shows the peak fast memory consumption and the fast memory size for

(a) Memory Overhead (b) Scalability.

Figure 6.8: Memory Overhead and Scalability of ReDL.

different ResNets. We use four different ResNets with different computation layers to illustrate ReDL's scalability. Figure 4.6(b) shows that with more layers added to ResNet, its peak memory consumption increases from 5.8G (ResNet 32) to 36GB (ResNet 152) and the fast memory size increases from 1.1GB to 7.35GB. In total, when increasing ResNet's layer from 32 to 152, the fast memory size rises slower than the peak memory. This behavior demonstrates the scalability in saving fast memory size and memory management effectiveness by using ReDL.

CHAPTER 7: UNIFIED HYBRID MEMORY SYSTEM FOR IN-MEMORY COMPUTING WITH DEEP LEARNING

7.1 Architecture Overview

In this section, we propose and implement UniRedl, a unified hybrid memory system based on NUMA running between applications and the backend deep learning frameworks.

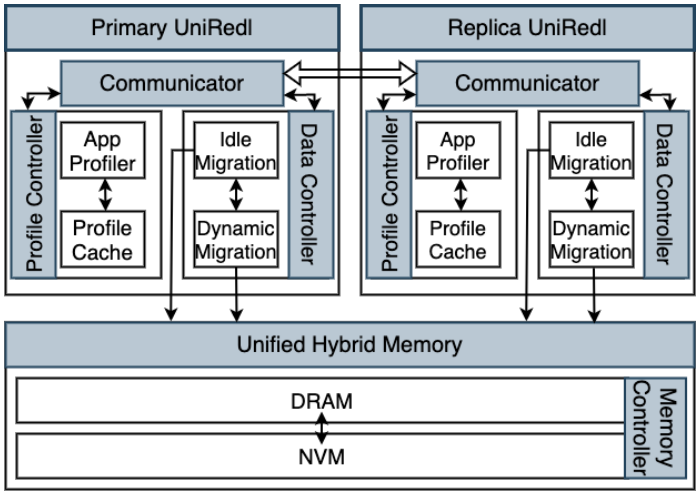


Figure 7.1: Architecture Overview of UniRedl.

Figure 7.1 depicts an overview of our proposed architecture. UniRedl adopts a primary/replica mode, one node runs the primary UniRedl process while other nodes are deployed with replica UniRedl processes. In UniRedl, the DRAM and NVM are abstracted as a unified memory layer. All the data placement and migration information is saved into the primary UniRedl node and then synchronized to the replica UniRedl nodes. Moreover, when deep learning applications are launched, their computation graphs as well as data access patterns are first profiled. This information is also distributed among all nodes so that the requested data can be proactively and timely moved to the right node where the application is running. As Figure 2.4 shows, our proposed UniRedl architecture mainly consists of

four components: *Profile Controller*, *Data Controller*, *Communicator*, and *Memory Controller*.

Profile Controller is designed to profile the computation graph and data access pattern of deep learning application, get the uniform data access demands of big data workloads. Two functional models are included in this components: App Profiler and Profile Cache. The App Profile builds the computation graph, collects memory access pattern of each kernel in computation graph, records access time of each data, and calculate the execution time of each kernel for deep learning applications. Moreover, it also collects the uniform data access behaviors of big data applications. The profiling step is simple and only takes one training loop and then updates all the information into the Profile Cache module. The Profile Cache is a preserved cache to keep profiled information on applications. Since profiling step may be time-consuming, the Profile Cache is necessary especially with the facts that deep learning applications typically have many static computation graphs and big data workloads usually contain uniform data access patterns. Besides, the contents of cache is periodically shared and synchronized across all the UniRedl nodes with RPC protocol.

Data Controller is driven by the Profile Controller component and implements the smart NUMA-based hybrid memory management strategy we proposed. Hot data and frequent-accessed data are managed by Unified Idle Migration module while the other normal data is controlled by Unified Dynamic Migration module. Both of the migration modules directly interact with the Unified Hybrid Memory via the interfaces provided by the *Memory Controller* component. Moreover, some metric for memory management such as data access information, memory usage, and memory bandwidth utilization are updated by the two migration modules and notify the hybrid memory layer when memory tension occurs. The smart NUMA-based memory management strategy take the above metrics into consideration when launches data placement and migration among the hybrid memory layer. We also introduce proactively data migration and direct memory access mechanisms to *Data Controller* among nodes to handle the case that some data is not timely migrated to the

right place.

Communicator takes the communication role among all UniRedl nodes. Each node is deployed a *Communicator* component to transmit and synchronize data and control information. When the Profile Controller running in the primary UniRedl node finishes profiling step and flushes all the information into its Profile Cache, the local Communication component is invoked to synchronize the content of cache to other replica UniRedl nodes. Similarly, once some changes occur in replica nodes such as data locality, application deployed location, local memory usage, and memory bandwidth utilization, the updated information will be collected by the Data Controller component and then sent back the primary node. Thus the remaining replica nodes are notified by the primary node to launch synchronizations. The communiations managed by the Communication component are built with periodical RPCs which guarantee stable connections and latest cache information.

Memory Controller abstracts the DRAM and NVM as a unified hybrid memory, which is shared among all the UniRedl nodes. It offers a huge memory space to each node and unified storage management. This component also provides some general interfaces e.g., memory allocation, data migration, direct data access, and memory utilization to the modules in Data Controller. When a memory tension or memory allocation request is initiated by the UniRedl node, the Memory Controller can receive the massage and response the request via migrating some unnecessary data from DRAM to NVM or allocate new memory space. This operation is committed in background and try not to affect the running of any applications so as to bring not harm to performance. Beside, the data placement and migration among DRAM and NVM overlap the computation process to further relief the side effects of waiting for requested data. The new location of data will be sent back to Data Controller when data migration is done, which make sure the information across all the UniRedl nodes are up to data.

7.2 Memory Allocation Strategy

The memory allocation strategy aims to improve the performance of deep learning and big data applications in hybrid memory system. Considering the inherent features in DRAM and NVM, the different accessing latencies of the DRAM and NVM are much higher than the overhead of NUMA-based hybrid memory systems. A smart data placement and migration strategy is necessary. Since we design a profiling module to determine the initial location of data as well as data access pattern, the smart memory allocation strategy can bring a lower memory access latency with higher bandwidth utilization both in DRAM and NVM. UniRedl provides a new memory allocation strategy named *HiLowAlloc*. Specifically, *HiLowAlloc* can minimize the average memory access latency while maximum the bandwidth utilization in DRAM and NVM simultaneously. The UniRedl provides some interfaces to implement the *HiLowAlloc* memory allocation strategy. The applications does not need do too many changes to the original source code with just a couple line to invoke the new interfaces.

The execution time of applications running in the hybrid memory system are basically determined by migrating data from the memory layer to computing units [23]. In our case, we assume that the execution time of an application is mainly decided by the total memory access overhead in the hybrid memory layer. Given that the DRAM and NVM memory are managed with a logical address space, the problem of optimizing execution time is converted to finding an optimal ratio between DRAM and NVM.

We define the total memory access latency on DRAM and NVM as L_{dram} and L_{nvm}, the total data transferring time from DRAM and NVM to computing units as B_{dram} and B_{nvm}, so the objective function is formulated as follows:

$$max(\frac{L_{dram}}{L_{nvm}}, \frac{B_{dram}}{B_{nvm}}) \qquad (7.2.1)$$

Let N_{dram} and N_{nvm} represent the total access times of DRAM and NVM, respectively. T^r_{dram} denotes the read latency on DRAM. We only consider the read latency from DRAM

because most of the data transfer between the DRAM and computing units is referring to read data from DRAM. T^r_{nvm} and T^w_{nvm} express the asymmetric write and read latency on NVM. We also demote the possibilities of write and read operations on NVM are p and q, respectively. Besides, the numbers of data hosted on DRAN and NVM are N_{dram} and N_{nvm}. So the total memory access latency on DRAM can be expressed as:

$$L_{dram} = N_{dram} * T^r_{dram} \qquad (7.2.2)$$

And the total memory access latency on NVM is:

$$L_{nvm} = N_{dram} * (p * T^r_{dram} + q * T^w_{dram}) \qquad (7.2.3)$$

Considering that UniRedl can profile the computation graph and data access pattern in deep learning applications and collect the uniform data access behaviors in big data workloads, the necessary data is supposed to place on the right node where the associated application is deployed. In this case some of inter-node communication and data transfer cost is saved. As a result, the ratio of DRAM and NVM determines the execution time of applications. Moreover, the total number of data placed on DRAM and NVM are equal, which means $N_{dram} = N_{nvm}$. According to Equation 7.2.2, 7.2.3, we can formulate the ratio of DRAM and NVM as:

$$\frac{L_{dram}}{L_{nvm}} = \frac{T^r_{dram}}{p*T^r_{dram}+q*T^w_{dram}} \qquad (7.2.4)$$

Similarly, the total data transferring time from DRAM and NVM to computing units as B_{dram} and B_{nvm} can be expressed as

$$\frac{B_{dram}}{B_{nvm}} = \frac{T^r_{dram}}{p*T^r_{dram}+q*T^w_{dram}} \qquad (7.2.5)$$

Based on Equation 7.2.1, we take the large ratio between $\frac{L_{dram}}{L_{nvm}}$ and $\frac{B_{dram}}{B_{nvm}}$ as the optimal ratio to guarantee the minimum application execution time with lower memory access latency and higher memory bandwidth utilization.

In hybrid memory system, both DRAM and NVM are abstracted as a union and unified in a logical memory address, but the hardwares of DRAM and NVM are still installed on

each physical node. The capacity of local memory is still limited. Although each node can direct access the memory of the other, the latency is higher compared with accessing data saved into the local memory. To tackle this problem, *HiLowAlloc* also the capacity of local memory and communication cost among nodes into account when doing data placement. UniRedl profiles the computation graph and data access pattern of deep learning and big data applications, respectively. Computation are committed sequentially in the static computation graph and data access behavior in the big data application is uniform. Therefore, there is not data migration in the training step. We can formulate the objective function as:

$$min \sum_{s\in\epsilon, m\in\kappa} \rho_{s,m} \qquad (7.2.6)$$

$$\text{s.t.} \begin{cases} \sum_{m\in\kappa} f_{(s,m)} + \sum_{m\in\kappa} s_{(s,m)} \geq \kappa & (1) \\ \sum_{m\in\kappa} f_{(s,m)} \leq DS & (2) \\ \sum_{m\in\kappa} s_{(s,m)} \leq NS & (3) \\ \sum_{m\in\kappa_(DS)} c_{(s,m)} \leq \sum_{m\in\kappa_(NS)} c_{(s,m)} & (4) \end{cases}$$

where *s* is a stage in the application and $\rho_{s,m}$ is the expected execution time of stage *s* under the data access behavior *m*. ϵ and κ denote the full application stage set and memory access pattern, respectively. Note that $\rho_{s,m}$ is highly determined by the location of data placement. More data placed into the DRAM brings better execution time while the data transfer between DRAM and NVM may hurt the performance. Therefore, one of the possible solution is placing data requested by the next computation stage and overlapping data transfer with current computation following the Constraints (1)(2)(3)(4). Constraint (1) denotes the total memory usage in DRAM and NVM is no smaller than the total size of local memory. Constraints (2) and (3) represent the memory space used in DRAM and NVM memory, respectively. Constraints (4) denotes the communication cost among local and remote DRAM is less than local DRAM and local NVM.

To solve the objective function Equation (7.2.6), we adopt the linear and integer programming theory [99], which uses the profiling information to automatically optimize data

placement with memory size and data transfer cost as the constraints.

In UniRedl, the *HiLowAlloc* memory allocation strategy is mainly divided into four steps. (1) The primary UniRedl node profiles the data access pattern, computation graph, and application deployment location of applications. All the profiled information is saved in the profile cache and then synchronized to the replica UniRedl ndoes when the profiling step is done. (2) The ratio of DRAM and NVM is calculated based on the above equations and profiling information following the *HiLowAlloc* memory allocation strategy. If the available DRAM space in a node is not enough to hold the requested data, partial of data is to be placed into the NVM instead. Data migrations occur with application runs on, when some data is paged out from the DRAM, the requested data is proactively moved into DRAM from NVM to save time spent in waiting for the required data. (3) The target replica UniRedl nodes are selected based on the ratio obtained in Step 2, which offer adequate free memory space and the minimum inter-node communication cost. These node also guarantee a lower intra-node communication cost by less data migrations between DRAM and NVM, especially when the application is running. (4) The data is to be placed on DRAM and NVM on the selected nodes based on the optimized results from Equation (7.2.6). If there is not enough free space in DRAM on a node, its pre-assigned NVM will hold the data. If the whole hybrid memory layer is full, UniRedl swaps data into the local hard disk to reclaim the space in memory.

Algorithm 6 depicts the main procedure of the *HiLowAlloc*, mainly including *ProfileApp*, *CalDNratio*, *ChooseNode*, and *placeData* functions. The four functions correspond to the four steps we mentioned above. *ProfileApp* (Line 1-6) in the primary UniRedl node profiles all the data and computation graph information and synchronizes to the replica UniRedl nodes. Then *CalDNratio* (Line 7-14) is invoked to calculate the optimal DRAM-to-NVM ratio based on Equation 7.2.1. According to the ratio, some potential target node is selected via function *ChooseNode* (Line 15-18). And then the last step in *HiLowAlloc* is called with data is finally placed on the rightly selected nodes, which is implemented by the *place-*

Algorithm 6 HiLowAlloc Memory Allocation

Input: app_id, mem_addr, $node\_id$
Output: $data\_info$, mem_info

```
1:  function PROFILEAPP(app_id)
2:      if app_id is DL app
3:          data_info, graph_info = profile(app_id)
4:      else data_info = profile(app_id)
5:      return data_info, graph_info and upDate Profile Cache
6:  end function
7:  function CALDNRATIO(data_info, graph_info)
8:      latency_ratio = calLatency(data_info)
9:      bandwidth_ratio = calBandwidth(data_info)
10:     if latency_ratio > bandwidth_ratio
11:         opt_ratio = latency_ratio
12:     else opt_ratio = bandwidth_ratio
13:     return opt_ratio
14: end function
15: function CHOOSENODE(opt_ratio, node_id)
16:     if checkNode(opt_ratio, node_id) is TRUE
17:     return node_id
18: end function
19: function PLACEDATA(opt_ratio, node_id)
20:     if mem_addr in DRAM is empty or enough space
21:         mem_addr = placeData(opt_ratio)
22:     else placeData(mem_n vm_a ddr)
23:     if all is full, freeVictimData()
24:     return mem_addr, data_info
25: end function
26: function UPDATEINFO(data_info)
27:     return data_info = updateInfo(mem_addr,data_info)
28: end function
```

Data function ((Line 22). The *HiLowAlloc* memory allocation strategy tackles the issue of placing data in a hybrid memory system. According to the profiling information, the required data can be placed into the right node where application is running, which improves the data locality as well. *HiLowAlloc* avoids high inter-node communication cost between DRAM and NVM by proactively migrating more requested data into DRAM and efficiently overlapping data transfer with computation. Intra-node communication cost is decreased with the mechanism that places data as close as possivbe to computing unit. Besides,

lower memory access latency and higher bandwidth utilization is achieved with the optimal DRAM-to-NVM ratio. The application execution time is reduced with higher memory utilization and less communication cost. When a data placement request is successfully committed, *updateInfo* (Line 26-28) is invoked to send all the updated information to the primary node.

7.3 Data Migration

We design two data migration strategies in UniRedl: *Unified Idle Migration* and *Unified Dynamic Migration* to manage hybrid memory. Specifically, *Unified Idle Migration* aims to manage data placed in DRAM while *UnifiedDynamic Migration* manages data saved in NVM. In this section, we present the details on how each migration strategy work. All the details are abstracted as interfaces to the applications running in hybrid memory system, which makes DRAM and NVM as a unified memory and easy to use.

7.3.1 Unified Idle Migration

Unified Idle Migration manages hot and frequent access data in DRAM aiming to guarantee a smooth application process by reducing unnecessary suspense caused by data missing. During the profiling step, the frequently accessed data is labeled as hot data along with the computation stages request it. A continuous memory space is allocated in DRAM to host hot data. This space is reserved for hot data and frequent access data. The data placed in DRAM is used to guarantee a better application execution time and it will not be easily migrated out if memory tension occurred in DRAM. When a data is requested and its not in the DRAM, replica UniRedl first check its profile cache to locate the data. If the data is in DRAM of other node, the primary UniRedl node approves the direct access request to the target data in other node and send a copy of this data to the local replica node. If the demanded data is in NVM, no matter it is in the local or remote NVM, the data is transferred to the DRAM where application is running. To save the communication and transfer cost, the data migration between DRAM and NVM are committed in advance with

the guidance of computation graph. Meanwhile, the data access information and memory utilization (such as available capacity, bandwidth utilization, data locality) is collected during application running in order to keep the information saved in profile cache is up to date.

Algorithm 7 Unified Idle Migration

Input: $data\_info$, mem_addr, app_id, $node\_id$
Output: $data\_info$, mem_info

```
1: function AllocDramMem(mem_addr)
2:     if free_size > getDramSize(mem_addr)
3:         mem_info = DramMalloc(mem_addr)
4:         free_size--
5:     return mem_info
6: end function
7: function ParseProf(data_info)
8:     graph_info = parseInfo(data_info)
9:     return graph_info
10: end function
11: function DramPlacement(mem_addr, data_info, graph_info)
12:     if mem_addr is empty or enough space
13:         mem_addr = placeData(data_info)
14:     else freeVictimData()
15:     return mem_addr, data_info
16: end function
17: function DramMigration(mem_addr, data_info, graph_info)
18:     if data_info is in remote DRAM
19:         mem_addr = migrateData(data_info, node_id)
20:     if data_info is in local NVM
21:         mem_addr = migrateData(data_info)
22:     if data_info is in remote NVM
23:         mem_addr = migrateData(data_info, node_id)
24:     return mem_addr, data_info
25: end function
26: function UpdateInfo(data_info, mem_addr)
27:     return data_info = updateInfo(mem_addr,data_info)
28: end function
```

Algorithm 7 depicts the main procedures of *Unified Idle Migration*, mainly including, *AllocDramMem*, *ParseProf*, *DramPlacement*, *DramMigration*, and *UpdateInfo* functions. (1) Some DRAM space is allocated when the application starts to run. *AllocDramMem*

(Line 1-6) finishes the application for a continuous memory space in DRAM if the free space satisfies the requested memory size. (2) Then the *ParseProf* (Line 7-10) function parses the profiling files and obtains the data access information and computation graph. Based in the profiles, UniRedl determines which node to place the data, which memory media to hold the data and which data to prefetch in the next stage. (3) When a new data request arrives and it is not loaded in the DRAM, *DramPlacement* (Line 11-16) is invoked. If the DRAM is empty or it has enough free space to host the new data, then data placement is committed. Otherwise, some victim data is selected and then moved out. The victim is the data would not be accessed in the current application running period based on the computation graph. (4) Data migration is operated with *DramMigration* (Line 17-25). To avoid the memory hit missing occurs, which the demanded data is not found in local DRAM(We define the node where application is running as local node and others as remote node, DRAM and NVM also follow this rule), UniRedl first search the profile cache to check if the requested data is in the DRAM of other node. If so, the primary UniRedl approves the direct memory access among replica nodes. Meanwhile, a copy of this data is sent to the local node. If the requested data is saved into the local NVM in the node where application is running, the data is to be migrated to DRAM. We introduce this mechanism that the priority of using the remote DRAM is higher than using the local NVM, because the latency of access from local NVM is much higher than the one from remote DRAM. There is another case that the requested data is saved in the remote NVM. UniRedl adopts the direct copy strategy to transfer the requested data from remote NVM to local DRAM. Besides, UniRedl tries to commit data migration in a proactive way that overlaps with the current computation process so that the running of application will not be interrupted. (5) The *UpdateInfo* (Line 26-38) function updates the metric information of the data and memory. And this information helps make data management decisions in the next computation period.

7.3.2 Unified Dynamic Migration

Unified Dynamic Migration controls data migrations in NVM. It uses the scalable migration window to determine the amount of data to be placed into NVM. The critical procedure in Unified Dynamic Migration strategy is find the optimal migration window that brings better application execution time with less suspense caused by memory missing hits. In UniRedl, the application running step is divided into a couple of migration windows. Since the deep learning applications have a stable computation graph and runs on with layer-by-layer, we use the layers as the metrology to decide the migration windows. Besides, the big data applications have the uniform data access patterns, which makes the layer-based migration window division applicable. The layer-based migration window mechanism usually guarantee the completion of each stage in applications.

The basic concern of Unified Dynamic Migration is timely migrating requested data to the computation in the next migration window from NVM to DRAM at the end of each migration windows. Given that the computation is to be finished at the end of migration window and the demanded data for the coming computing can be loaded from profile cache, proactively locate the data and transfer it into the right DRAM is possible. During the data migration procedure, memory tension may occur in DRAM because of limited capacity. To handle this issue, Unified Dynamic Migration migrates the data that has not been accessed back to NVM in order to save the DRAM space. At the beginning of migration window, all the requested data is already in DRAM to guarantee a smooth computation stage. Computation and data migration are overlapped in each migration window, which ensures better application execution time and higher memory utilization.

Another case to consider is the migration window may fail to satisfy some of the computation phases due to there is not enough time to finish migrating the requested data from NVM to DRAM before the upcoming window begins. To solve this problem, we introduce direct NVM access strategy, which breaks the barrier between computing units and NVM by allowing computing units to directly access data placed in NVM. In the regular

scenario, computing units such as CPU, GPU, and FPGA directly access the data saved in NVM. Data saved in NVM is first migrated in DRAM and then computing units can use them. With direct NVM access, computing units can directly access data in NVM by skipping the intervene of DRAM. Application performance from direct NVM access cannot compete with the DRAM access mode, but it does not occur frequently and its adverse impacts are negligible.

Determining an optimal migration window is a difficult and dilemma. If the adopted migration window is too large, then the total number of data objects to be migrated may be even larger than the free DRAM space. If the migration window is too small, the possible execution time to overlap with computation is shorten. Besides, the requested data for the coming computation may not be timely migrated to the right place before the next migration window begins. To tackle this problem, we adopt the iterated greedy algorithm [100] to find the optimal migration window at runtime.

Based on the profiled data access pattern and computation graph, we start with the migration window of the median of the total computation layers/stages and then test if this windows size brings better application execution time and higher memory utilization. If the test returns a positive result, the optimal migration window is determined. Otherwise, we will first repeat this process with a new migration window by adding 1 layer to the current size. And then test the case by deducting 1 layer from the median number of total layers. Three basic training iterations are covered in one optimizing round. The next round is committed with the case that the migration windows are added or deducted 2 to the median number. The similar round is repeated until the optimal migration window is determined. We collect the application performance from different migration windows and select the best one as the migration window for the coming computation. Given the fact that the deep learning applications perform repetitive execution and the big data applications have uniform data access pattern, data placement and migration in any optimizing round is the same in order to guarantee the correctness of comparison results. Although it may take a couple

of rounds to determine the optimal migration window and bring some performance loss, the total overhead is not large because this case does not often happen and performance loss is compensated in the remaining computations.

Algorithm 8 Unified Dynamic Migration

Input: $data\_info$, mem_addr, app_id, $node\_id$
Output: $data\_info$, mem_info

```
1:  function AllocNvmMem(mem_addr)
2:      if free_size > getDramSize(mem_addr)
3:          mem_info = NvmMalloc(mem_addr)
4:          free_size--
5:      return mem_info
6:  end function
7:  function ParseProf(data_info)
8:      graph_info = parseInfo(data_info)
9:      return graph_info
10: end function
11: function FindOptWindow(data_info, graph_info)
12:     opt_period = findPerid(data_info, graph_info)
13:     return opt_period
14: end function
15: function NvmMigration(opt_period, mem_addr, data_info)
16:     if mem_addr is empty or enough space
17:         mem_addr = placeData(data_info)
18:     else freeVictimData()
19:     return mem_addr, data_info
20:     //Data Migrated to DRAM
21:     if opt_period is valid
22:         mem_addr = migrateData(data_info)
23:     else dirctAccessNvm(mem_pts)
24: end function
25: function UpdateInfo(data_info, mem_addr)
26:     data_info = updateInfo(mem_addr,data_info)
27:     return data_info
28: end function
```

Algorithm 8 depicts the main procedures of *Unified Dynamic Migration*. *AllocNvmMem*, *ParseProf*, *FindOptWindow*, *NvmMigration*, and *UpdateInfo*. (1) *AllocNvmMem* (Line 1-6) implements the space allocation in NVM when applications starts to run. A continuous NVM memory space is also necessary to reduce the overhead in locating data. (2) The

ParseProf (Line 7-10) function parses the profiling files and obtains the data access information and computation graph. According to the profiling information, UniRedl then determines the node to place data. (3) We implement the proposed iterated greedy algorithm in *FindOptWindow* (Line 11-14), aiming to find the optimal migration windows. This procedure is actually a part of application execution so that it will not affect the processing of the application. As discussed in the above context, determining the optimal migration window might take a couple of rounds and cause some performance loss, but this loss can be made up in the coming computation by reducing data migration and increasing memory utilization. (4) *NvmMigration* (Line 15-24) implements data migration. When a data migration request arrive to *NvmMigration*, the current replica UniRedl node first checks if it is a local or remote data migration request. For the local migration request, *NvmMigration* ensues the demanded data is migrated to DRAM within the current migration window. For the remote request, the replica UniRedl node first contact the primary node for approval. Once received the approval, the replica node releases the demanded data and then transfer to the destination node. When there is no free space in NVM, *NvmMigration* migrates some data based on the access information and computation graph to the local disk. Besides, We implement direct NVM access in *accessSlowMem* function (Line 23) in *NvmMigration* to handle the case that data migration to DRAM cannot be finished due to the migration period does not fit to some computations. With direct NVM access, the computation continues by directly accessing data saved in NVM and does not need to wait for the completion of data migration to DRAM. (5) The *UpdateInfo* (Line 25-28) function updates the metric information of the data and memory. And this information helps make data management decisions in the next computation period.

In UniRedl, all the data are uniformly managed with idle migration and dynamic migration in the hybrid memory system. The idle migration policy tackles the issue of hosting hot data in DRAM even if it is no longer accessed. Unified Idle Migration decreases unnecessary data migrations between DRAM and NVM, and thus improves application

performance and memory bandwidth. The idle migration policy controls data migrations to DRAM, and offers direct NVM access to remote node. Idle migration and dynamic migration together bring better application execution time, memory utilization and lower communication cost.

7.3.3 Data Migration Workflow

Figure 7.2 depicts an example of the data access workflow in UniRedl in a logical view. Three UniRedl nodes are included: one primary node and two replica nodes.

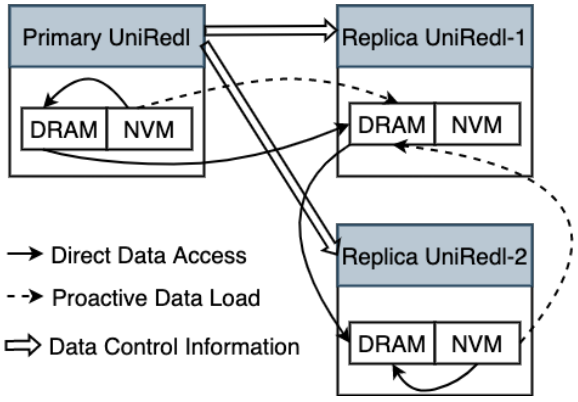


Figure 7.2: Logical workflow of data access in UniRedl.

In Figure 7.2, we assume that all the profiling information has been synchronized and up to data. Replica UniRedl Node 1 is the node where the application is running. The solid and dotted arrow lines represent direct DRAM data access and proactively data transfer from local/remote NVM to DRAM. The hollow arrow denotes the control information such as, profiling information, data access request, and memory utilization, among nodes. In Figure 7.2, each node has its local DRAM and NVM, but all the memory media is logically connected and uniformly managed by UniRedl. The physically separated mode make it easy to describe the workflow and interactions among nodes. We define the node where application is running as local node and others as remote node, DRAM and NVM also follow this rule.

In Figure 7.2, the data for application running in replica UniRedl Node 1 is placed in both DRAM and NVM memory according to the profiling information by primary UniRedl

node. The application directly load demanded data from its local DRAM. When memory missing occurs, replica UniRedl Node 1 first check its profile cache to locate the location of demanded data. There are four possible locations: (1) If the data is saved in DRAM of a remote node, as the replica UniRedl Node 2 listed in this figure, Node 1 then sends a remote direct DRAM data access to primary UniRedl node. Once the primary node approves this access remote, it will notify Node 1 and 2. And then Node 2 starts the remote direct DRAM data access and a copy of this data is sent to Node 1 simultaneously. The purpose of data copy is to ensure the next request on this data is accessed in local DRAM. This operation is done by idle migration policy. (2) If the data is found in the local NVM of Node 1, he data is to be migrated to the local DRAM. This procedure is committed with dynamic migration strategy. We adopt this mechanism that the priority of using remote DRAM is higher than using local NVM, because the latency of access from local NVM is higher than the one from remote DRAM. (3) If the data is in a remote NVM, as showed in Figure 7.2, the data Node 1 requests is saved in the primary mode. Node 2 will send a remote direct NVM data access to the primary node. With the approval message from primary node, Node 2 can access the data saved in the NVM of primary node. (4) If the data is not found in any DRAM or NVM of any node, it is saved in a local disk. In this case, dynamic migration strategy locates and transfers the requested data to the DRAM of Node 1. Moreover, all the data migrations are committed in the optimal migration window, in which migrations overlap with computations so that the applications does not need to suspend.

7.4 Evaluation

7.4.1 Experiment Setup

We study the performance of UniRedl on a 4-node cluster, which has Intel Xeon(R) CPU E5-2630v4@2.20GHz x 20 and 64GB DDR4 DRAM. All the nodes are connected with 1Gb ethernet network adapters. We use Persistent Memory Block Driver (PMBD) [53] to simulate hybrid DRAM and NVM on NUMA-based system. Specifically, 16GB DRAM of each node is configured to emulate NVM. We configure the read/write latencies of NVM

Table 7.1: Summary of selective benchmarks.

Benchmark	Dataset	Batch size	Model size
Terasort	TeraGen	N/A	N/A
Kmeans	GaitData	N/A	N/A
ResNet 152	CIFAR-10	128	232 MB
VGG 19	CIFAR-10	64	539 MB

to be 3X and 8X of the DRAM, respectively. And the bandwidth of NVM is limited to be half the DRAM's bandwidth. These configurations are very close to the specifications of commercial Intel Optane DC Persistence Memory [35]. The nodes run Ubuntu 16.04 LTS operating system with kernel version 4.0, Spark 2.6 and TensorFlow 2.0. We adopt Intel Memory Latency Checker (MCL) [57] to quantify the communication overhead of inter/intra-nodes and memory utilization.

We select five workloads to benchmark our work. Terasort [55], Kmeans [104] are two typical big data applications while ResNet152 [56], and VGG 19 [105] are deep learning workloads. To evaluate the performance of workloads, we run big data and deep learning applications on Spark and TensorFlow, respectively. For deep learning applications, we set its inter-op parallelism and intra-op parallelism to be 20 to ensure this parameter is consistent to the physical cores in our testbed, so that all the experiments are conducted on the CPU with one thread running on each physical core. All workloads run until their execution time in a migration window is stable. Since all the workloads have no data dependent relations and their data access patterns are predictable, the performance will tend to be stable after a round of computation iterations. A brief summary of the workloads is illustrated in Table 7.1. Unless specified otherwise, all the experiments are conducted on a pure CPU platform. We will leave the CPU and GPU mixed computations in future work.

We use the traditional NUMA management and NUMA with the automatic NUMA balancing (anb) [54] as the baselines. We also compare the performance of UniRedl with BMPM [86] and OS-Integrated Multi-level memory management system (OIM) [21]. The performances of our work are mainly evaluated in (i) execution time, (ii) data locality, (iii)

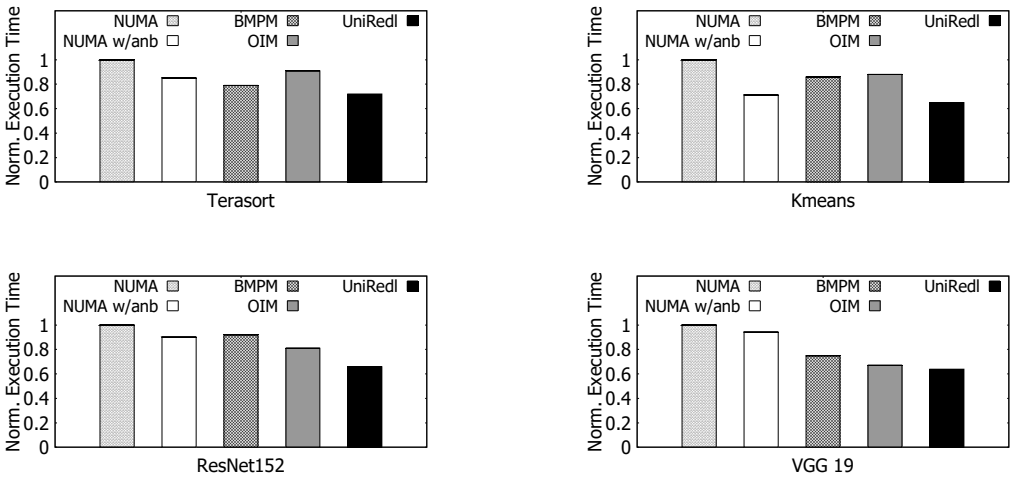


Figure 7.3: Execution time from NUMA, NUMA with anb, BMPM, OIM and UniRedl

bandwidth utilization, and (iv) overhead.

7.4.2 Execution Time

Figure 7.3 shows the execution time of workloads with different data management strategies in the hybrid memory system, all the experimental results are normalized to the baseline traditional NUMA. The figures depict that UniRedl brings the most performance improvement of 33.2% on average when compare to the traditional NUMA . Because the default memory management strategy adopted in traditional NUMA is preferentially placing data on DRAM and then DVM without considering the asymmetrical bandwidth. OIM has fluctuating impacts on the performance improvement due to it implements data migration in the unit of page by launching 4 threads for paralleling page copy and 8 threads for concurrently migrating pages between DRAM and NVM, which may not efficient for some latency-sensitive workloads, such as Kmeans and ResNet152. In contrast, UniRedl reduces the overall execution time of these workloads by 26.1% and 18.5%, respectively. BMPM adopts a memory bandwidth-awareness page management strategy, and it achieves an average 17.6% performance improvement over the traditional NUMA. The performance improvement that brought by BMPM on ResNet152 is not so obvious. We can explain that the ResNet152 is a latency-sensitive workload and BMPM does not take this part into

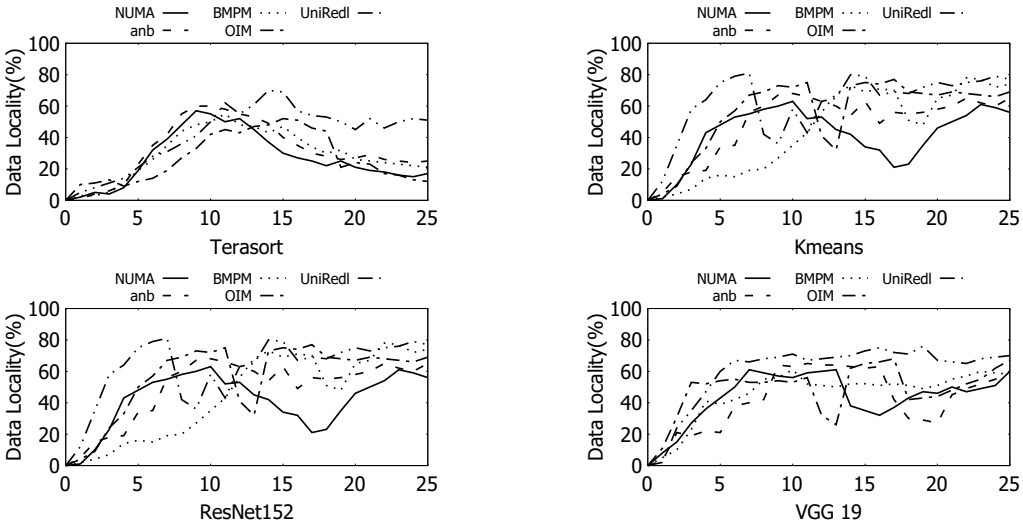


Figure 7.4: The tendencies of Data Locality in DRAM from NUMA, NUMA with anb, BMPM, OIM and UniRedl

consideration when committing data management while UniRedl is specially designed for latency-sensitive workloads. VGG 19 obtains the most improvement in execution time by 36%, 31.8% from UniRedl and NUMA-anb. The improvement from OIM is not so huge with about 0.4% due to the coarse page management policy. Overall, our proposed method UniRedl averagely improves application performance by 33.2%, 20.6%, 19.0%, and 17.5% compared to the traditional NUMA, NUMA with anb, BMPM, and OIM, respectively.

7.4.3 Data Locality

We also evaluate data locality in DRAM of the five data management strategies. We define the percentage of successful data access in DRAM as data locality. A higher data locality rate means less data migrations between DRAM and NVM, and then reflects the efficiency of data management strategy. Figure 7.4 shows the tendencies of data locality within a limited time window, which reflects a 25-second behavior when the workloads start to run. We can observe that all the data placement strategies demonstrate fluctuations in data locality and will come to a relatively stable status. Furthermore, each fluctuation displays a data migration operation within a migration window. The valleys between any two consecutive peaks illustrate data migration. UniRedl achieves 52.0% , 34.3%, 30.6%,

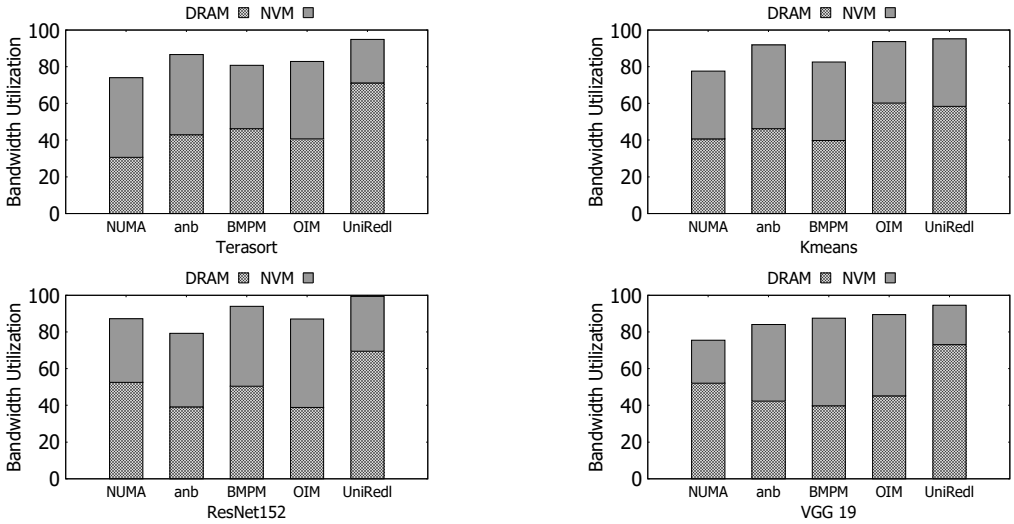


Figure 7.5: Bandwidth Utilization from NUMA, NUMA with anb, BMPM, OIM and UniRedl

22.1% on average over NUMA, NUMA with anb, BMPM, OIM, respectively. UniRedl also delays and decreases the occurrence of valleys by wisely and proactively committing data migration according to the profiled data access pattern and computation graph. OIM brings a little performance improvement in Terasort with about 0.05% over the traditional NUMA due to its coarse granularity management unit in data migration. For the VGG 19 workload, UniRedl offers the overall data locality with 59.8%, while other data management strategies bring performance less than 50%. There is an outlier in the experimental curve of VGG with the traditional NUMA; a valley appears around the time point 15. This exception is caused by a large convolution layer , which demands more memory while the default memory management policy of NUMA is not so efficient.

7.4.4 Bandwidth Utilization

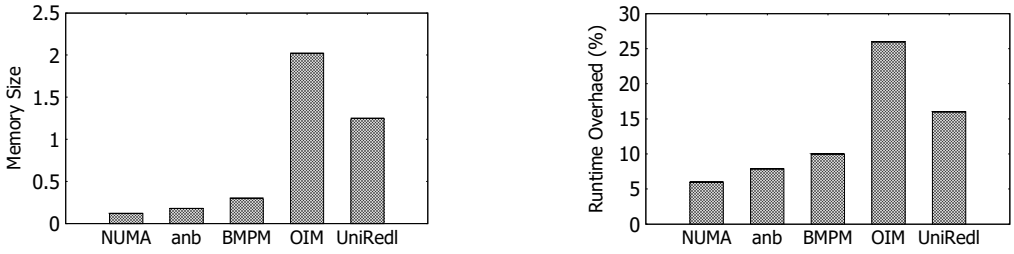
Figure 7.5 depicts comparisons on total memory bandwidth utilizations in DRAM and NVM. UniRedl achieves 22.2%, 12.4%, 11.4%, 8.8% by average improvements in total bandwidth utilization over NUMA, NUMA with anb, BMPM, and OIM, respectively. The traditional NUMA management strategy performs better in deep learning applications (ResNet152 and VGG 19) than big data applications (Terasort and Kmeans), because the

uniform data access pattern fails to take full advantages of the lower latency characteristic of DRAM. UniRedl brings a higher bandwidth utilization except Kmeans. In contrast, the NUMA-anb and OIM provide a better performance. The reason is that the dataset for Kmeans requests more migrations between DRAM and NVM, which introduces smaller data migration windows and less overlaps with computation procedures. The balance strategy adopted by NUMA-anb evenly distributes data in DRAM and NVM, which in return decrease migrations. The multi-threads strategy used by OIM can place more requested data in DRAM, and thus its performance is close to UniRedl. VGG 19 has a extremely large second convolution layer, which demands more memory space to host its related data objects, the higher bandwidth utilization comes from UniRedl with up to 19% improvement in DRAM. UniRedl only brings 21.6% memory bandwidth utilization in NVM. Given that the size of demanded data is limited, a higher DRAM utilization demonstrates more data is hosted in DRAM which results in less data in NVM. Moreover, this behavior reveals the efficiency of UniRedl in data migration in terms of profiled data access pattern and computation graph. BMPM uses a bandwidth awareness data placement strategy.

7.4.5 Overhead

To analyze the overhead of UniRedl, we conduct experiments to measure average peak memory consumption in NUMA, NUMA with anb, BMPM, OIM and UniRedl in the system level, which scales how much memory resource it takes to conduct data management. In Figure 7.6(a), NUMA follows the classic recently access policy to conduct data migration while BMPM only moves the demanded data between DRAM and NVM. They only introduce 0.12 GB, 0.18 GB, and 0.3 GB memory overhead. UniRedl and OIM all use profiled data access information to determine data migration, and profiling step also consumes some memory, thus they increase memory overhead to 1.25 GB and 2.02 GB. The overall average memory overhead introduced by UniRedl is about 8.4% (about 1GB), which is negligible to the whole system-level memory (64GB in our testbed).

We also evaluate the overhead in runtime, which is the percentage in the total application



(a) Memory Overhead (b) Runtime Overhead.

Figure 7.6: Overhead comparison in memory size and runtime from NUMA, NUMA with anb, BMPM, OIM and UniRedl

runtime. Figure 7.6(b) shows the NUMA strategies leads less runtime overhead with 6% and 7%, respectively. BMPM brings 10% more overhead in runtime because its bandwidth awareness migration policy demands some processing cycles. OIM brings 16% runtime overhead by deploying multiple threads to conduct data migration. UniRedl consumes 10% and 6% more runtime overhead compared with OIM, BMPM due to it needs some extra time to finish profiling and optimizing steps. However, the overhead from runtime could be counteracted by the performance improvement obtained with accessing more data from DRAM and decreasing data migrations in hybrid memory system.

CHAPTER 8: CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

In this document, we present the state of the art of works in memory management for In-memory Computing with Hybrid Memory System (HMS). We have followed the addressed challenges to design shared memory and hybrid memory systems while maintaining memory utilization efficiency and reducing unnecessary data migration of underlying shared in-memory computing frameworks. To tackle imbalanced memory utilization issues among multiple Spark executors, we propose and develop iMlayer, a shared memory cache space, which is deployed between on-heap memory and local disk, to cache and manage intermediate data across multiple executors so that I/O operations can be decreased. We further implement online learning on some existing frameworks by proposing iDlaLayer. To obtain a better DNN model quality with less training cost, we introduce a novel data life aware updating strategy (DLA), which relies on combined data sample and considers training cost when deciding whether to perform a model updating action. To tackle the data migration problem on HMS for DNN applications, we propose a runtime system for HMS that automatically determines the optimal data migration window and exploits domain knowledge on DNN to decide data migrations between the fast and slow memories in HMS. To achieve a better performance in data migrations for DNN training, we introduce a reference distance and location based data management strategy (ReDL) that treats short-lived and long-lived data objects with Idle and Dynamic migration methods, respectively. Using ReDL, conducting DNN model trainings on HMS with a smaller fase memory space can achieve the performance close to the pure fast memory system. We further extend ReDL to a cluster level with developing UniRedl, a unified memory system across the whole cluster, which automatically optimizes data migration between DRAM

and NVM based on data access pattern and computation graph of applications. To obtain a better application performance, we provides a new memory allocation strategy named *HiLowAlloc*. We also design two data migration strategies in UniRedl: Unified Idle Migration and Unified Dynamic Migration to manage hybrid memory. Specifically, Unified Idle Migration aims to manage data placed in DRAM while Unified Dynamic Migration manages data saved in NVM. Furthermore, we conduct extensive evaluations of the proposed works and techniques through implementations and simulations on a testbed of local cluster. The preliminary results of evaluation are very promising.

8.2 Future Work

Our research mainly focus on memory management strategy in-memory computing with hybrid memory system. The emerging of new memory technologies has attracted more interesting research topics, such as near data processing (NDP), processing in memory (PIM), disaggregated computing. Our future work will be launched as follows:

(1) Unify the memory management over CPU and GPU. Most of the current hybrid memory systems are mainly designed for CPU computation and has less performance improvement for GPU-centric computation. ZeRO-Infinity [106] breaks the GPU memory limitation by leveraging GPU, CPU and NVM memory to host more parameter for deep learning applications. But it still needs the assistance of CPU to finish data migration. We plan to move further by designing a hybrid memory system that provides direct memory access to all the computation units attached to it. Besides, this work will exploit the possibility in efficient memory management on hybrid memory system without introducing new hardwares.

(2) Exploit disaggregated computing systems. A typical use of NVM in the cluster level is aggregating shareable memory space across all the nodes in the format of hybrid shared memory pool, for example Hotpot [107]. However, this work need a global awareness of the memory address and more time to maintain this universal memory pool. A new trend of using NVM is totally contrary to the shareable memory. This solution distributes

NVM in multiple nodes and shared by applications running in the cluster. Disaggregated NVM systems equips NVM in a few nodes with less computation abilities, which serve as a stroage-like node to the computation nodes. This solution can guarantee less network bandwidth by only conducting data migration among specific nodes. However, this topic is still in the primary stage and need more work in software and hardware support.

(3) Include new hardware on in-memory computing. Near data processing (NDP) and processing in memory (PIM) can significantly reduce the overhead from data movement especially for machine learning algorithms such as deep neural network (DNN) and convolutional neural network (CNN). Some products have adopted partial of PIM, such as Google's Tensor Processing Unit (TPU), Intel Xeon Phi Knights Landing series CPU, and NVIDIA tesla V100 GPU, andBut there still is no real commercial products on real NDP or PIM. In our future work, we plan to start with some simulations about PIM and then launch the solid experiments on real PIM hardware.

(4) Build a NVM optimizer for traditional applications. The NVM technologies have been widely promoted in big data applications, such as machine learning, computer vision, recommendation systems, and K-V store. But most of those frameworks are developed following the traditional DRAM and disk memory architecture, which is not efficient in hybrid memory systems. We plan to design a middleware layer running between the applications and computing frameworks to automatically optimize the original memory management strategies in applications, so that the new characteristics (data persistence, fast bandwidth, huge capacity) brought by hybrid memory system can be fully leveraged.

8.3 Publications

1. Wei Rang, Donglin Yang, Dazhao Cheng. "Unified Hybrid Memory System for Scalable Deep Learning Applications" under peer review.

2. Wei Rang, Donglin Yang, Zhimin Li, and Dazhao Cheng, "Scalable Data Management on Hybrid Memory System for Deep Neural Network Applications" IEEE In-

ternational Conference on Big Data (BigData), 2021.

3. Wei Rang, Donglin Yang, Dazhao Cheng. "A Shared Memory Cache Layer across Multiple Executors in Apache Spark" IEEE International Conference on Big Data (BigData), 2020.

4. Wei Rang, Donglin Yang, Dazhao Cheng, Kun Suo, Wei Chen. "Data Life Aware Model Updating Strategy for Stream-based Online Deep Learning" IEEE International Conference on Cluster (Cluster), 2020.

5. Donglin Yang, Wei Rang and Dazhao Cheng. "Elastic Executor Provisioning for Iterative Workloads on Apache Spark" IEEE International Conference on Big Data (BigData), 2019.

6. Wei Rang, Donglin Yang and Dazhao Cheng. "Data Life Aware Model Updating Strategy for Stream-based Online Deep Learning" IEEE Transactions on Parallel and Distributed Systems (TPDS).

7. Wei Rang, Donglin Yang and Dazhao Cheng. "Dependency-aware Tensor Scheduler for Industrial AI Applications" IEEE Industrial Electronics Magazine (IEM).

REFERENCES

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets.," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[2] M. H. Iqbal and T. R. Soomro, "Big data analysis: Apache storm perspective," *International journal of computer trends and technology*, vol. 19, no. 1, pp. 9–14, 2015.

[3] M. Eriksson, "Monitoring, modelling and identification of data center servers," 2018.

[4] Z. Liu and T. E. Ng, "Leaky buffer: A novel abstraction for relieving memory pressure from cluster data processing frameworks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 128–140, 2017.

[5] H.-N. Vießmann, A. Šinkarovs, and S.-B. Scholz, "Extended memory reuse: An optimisation for reducing memory allocations," in *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, pp. 107–118, 2018.

[6] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "Memtune: Dynamic memory management for in-memory data analytic platforms," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 383–392, IEEE, 2016.

[7] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM computing surveys (CSUR)*, vol. 46, no. 4, p. 44, 2014.

[8] L. Xiao, "Dual averaging methods for regularized stochastic learning and online optimization," *Journal of Machine Learning Research*, vol. 11, no. Oct, pp. 2543–2596, 2010.

[9] S. J. Prince and J. H. Elder, "Probabilistic linear discriminant analysis for inferences about identity," in *2007 IEEE 11th International Conference on Computer Vision*, pp. 1–8, IEEE, 2007.

[10] J. Mairal, F. Bach, J. Ponce, and G. Sapiro, "Online learning for matrix factorization and sparse coding," *Journal of Machine Learning Research*, vol. 11, no. Jan, pp. 19–60, 2010.

[11] B. Tamara, B. Nicholas, W. Andre, and W. Ashia, "Streaming variational bayes," NIPS, 2013.

[12] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, *et al.*, "Ad click prediction: a view from the trenches," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1222–1230, ACM, 2013.

[13] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, *et al.*, "Practical lessons from predicting clicks on ads at facebook," in *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pp. 1–9, ACM, 2014.

[14] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Transactions on Management Information Systems (TMIS)*, vol. 6, no. 4, p. 13, 2016.

[15] D. Tang, F. Wei, B. Qin, T. Liu, and M. Zhou, "Coooolll: A deep learning system for twitter sentiment classification," in *Proceedings of the 8th international workshop on semantic evaluation (SemEval 2014)*, pp. 208–212, 2014.

[16] L. Yang, J. Shi, B. Chern, and A. Feng, "Open sourcing tensorflowonspark: Distributed deep learning on big-data clusters," 2017.

[17] S. Kannan, Y. Ren, and A. Bhattacharjee, "Klocs: kernel-level object contexts for heterogeneous memory systems," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 65–78, 2021.

[18] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos: Os design for heterogeneous memory management in datacenter," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 521–534, 2017.

[19] D. Yang, H. Liu, H. Jin, and Y. Zhang, "Hmvisor: Dynamic hybrid memory management for virtual machines," *Science China Information Sciences*, vol. 64, no. 9, pp. 1–16, 2021.

[20] Q. Wang, Y. Lu, J. Li, and J. Shu, "Nap: A black-box approach to numa-aware persistent memory indexes," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Virtual*, 2021.

[21] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 331–345, 2019.

[22] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 631–644, 2017.

[23] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "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*, pp. 607–618, 2015.

[24] A. ChiachenChou and M. Qureshi, "Batman: Maximizing bandwidth utilization of hybrid memory systems," 2015.

[25] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, *et al.*, "Software-defined far memory in warehouse-scale computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 317–330, 2019.

[26] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 369–383, 2016.

[27] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing dram footprint with nvm in facebook," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–13, 2018.

[28] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, pp. 160–167, 2008.

[29] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic gpu memory management for training deep neural networks," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 41–53, 2018.

[30] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 1396–1401, IEEE, 2017.

[31] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[32] K. Wu, J. Ren, and D. Li, "Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 401–413, IEEE, 2018.

[33] T. Hirofuchi and R. Takano, "Raminate: Hypervisor-based virtualization for hybrid main memory systems," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 112–125, 2016.

[34] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: a holistic approach to memory placement on numa systems," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 381–394, 2013.

[35] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[36] Z. Duan, H. Liu, X. Liao, H. Jin, W. Jiang, and Y. Zhang, "Hinuma: Numa-aware data placement and migration in hybrid memory systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pp. 367–375, IEEE, 2019.

[37] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, *et al.*, "Wide & deep learning for recommender systems," in *Proceedings of the 1st workshop on deep learning for recommender systems*, pp. 7–10, 2016.

[38] G. A. DeepMind, "Mastering the real-time strategy game starcraft ii," 2019.

[39] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[40] H. Zhang, T. Xu, H. Li, S. Zhang, X. Wang, X. Huang, and D. N. Metaxas, "Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, pp. 5907–5915, 2017.

[41] A. Vaswani, Y. Zhao, V. Fossum, and D. Chiang, "Decoding with large-scale neural language models improves translation," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1387–1392, 2013.

[42] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[43] F. Romero, B. Braun, and D. Cheriton, "The tracar ratio: Selecting the right storage technology for active dataset-serving databases," *arXiv preprint arXiv:2006.14793*, 2020.

[44] X.-W. Chen and X. Lin, "Big data deep learning: challenges and perspectives," *IEEE access*, vol. 2, pp. 514–525, 2014.

[45] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pp. 41–51, IEEE, 2010.

[46] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[47] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, *et al.*, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal processing magazine*, vol. 29, 2012.

[48] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[49] "Google experiments with tensorflow on criteo dataset." `https://labs.criteo.com/2017/02/google-experiments-tensorflow-criteo-dataset/`. Accessed: 2018-6-28.

[50] H. M. Wallach, I. Murray, R. Salakhutdinov, and D. Mimno, "Evaluation methods for topic models," in *Proceedings of the 26th annual international conference on machine learning*, pp. 1105–1112, 2009.

[51] J. Ren, J. Luo, K. Wu, M. Zhang, and D. Li, "Sentinel: Runtime data management on heterogeneous main memorysystems for deep learning," *arXiv preprint arXiv:1909.05182*, 2019.

[52] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[53] F. Chen, M. P. Mesnier, and S. Hahn, "A protected block device for persistent memory," in *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, IEEE, 2014.

[54] C. Lameter, "Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors.," *Queue*, vol. 11, no. 7, pp. 40–51, 2013.

[55] W. Gao, J. Zhan, L. Wang, C. Luo, Z. Jia, D. Zheng, C. Zheng, X. He, H. Ye, H. Wang, *et al.*, "Data motif-based proxy benchmarks for big data and ai workloads," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 48–58, IEEE, 2018.

[56] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[57] "Intel memory latency checker v3.9a."

[58] A. Barai, G. Chennupati, N. Santhi, A.-H. A. Badawy, and S. Eidenbenz, "Modeling shared cache performance of openmp programs using reuse distance," *arXiv preprint arXiv:1907.12666*, 2019.

[59] Y. Yuan, Y. Shen, W. Li, D. Yu, L. Yan, and Y. Wang, "Pr-lru: A novel buffer replacement algorithm based on the probability of reference for flash memory," *IEEE Access*, vol. 5, pp. 12626–12634, 2017.

[60] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, "Computing in memory with spin-transfer torque magnetic ram," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 470–483, 2018.

[61] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 481–492, IEEE, 2017.

[62] N. Bobroff, P. Westerink, and L. Fong, "Active control of memory for java virtual machines and applications.," in *ICAC*, pp. 97–103, 2014.

[63] M. Khan, M. A. Laurenzanoy, J. Marsy, E. Hagersten, and D. Black-Schaffer, "Arep: Adaptive resource efficient prefetching for maximizing multicore performance," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 367–378, IEEE, 2015.

[64] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, ACM, 2014.

[65] "Project tungsten: Bringing apache spark closer to bare metal." `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`.

[66] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ACM, 2015.

[67] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem, "Ginseng: Market-driven memory allocation," in *ACM SIGPLAN Notices*, vol. 49, pp. 41–52, ACM, 2014.

[68] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li, *et al.*, "Bigdl: A distributed deep learning framework for big data," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 50–60, 2019.

[69] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems*, pp. 1–16, 2016.

[70] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential dataflow.," in *CIDR*, 2013.

[71] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 439–455, 2013.

[72] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Presented as part of the*, 2012.

[73] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul, "Secret: a model for analysis of the execution semantics of stream processing systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 232–243, 2010.

[74] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[75] M. H. Javed, X. Lu, and D. K. Panda, "Cutting the tail: designing high performance message brokers to reduce tail latencies in stream processing," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 223–233, IEEE, 2018.

[76] G. Cauwenberghs and T. Poggio, "Incremental and decremental support vector machine learning," in *Advances in neural information processing systems*, pp. 409–415, 2001.

[77] R. Klinkenberg, "Learning drifting concepts: Example selection vs. example weighting," *Intelligent data analysis*, vol. 8, no. 3, pp. 281–300, 2004.

[78] D. Crankshaw, *The Design and Implementation of Low-Latency Prediction Serving Systems*. PhD thesis, UC Berkeley, 2019.

[79] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan, "The missing piece in complex analytics: Low latency, scalable model management and serving with velox," *arXiv preprint arXiv:1409.3809*, 2014.

[80] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3d xpoint technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.

[81] D. W. Chang, G. Byun, H. Kim, M. Ahn, S. Ryu, N. S. Kim, and M. Schulte, "Reevaluating the latency claims of 3d stacked memories," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 657–662, IEEE, 2013.

[82] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, vol. 30, no. 1, pp. 143–143, 2010.

[83] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *arXiv preprint arXiv:1904.07162*, 2019.

[84] T. Hirofuchi and R. Takano, "The preliminary evaluation of a hypervisor-based virtualization mechanism for intel optane dc persistent memory module," *arXiv preprint arXiv:1907.12014*, 2019.

[85] K. Wu, Y. Huang, and D. Li, "Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2017.

[86] S. Yu, S. Park, and W. Baek, "Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems," in *Proceedings of the International Conference on Supercomputing*, pp. 1–10, 2017.

[87] Z. Yang, D. Jia, S. Ioannidis, N. Mi, and B. Sheng, "Intermediate data caching optimization for multi-stage and parallel big data frameworks," *arXiv preprint arXiv:1804.10563*, 2018.

[88] H. Inagaki, R. Kawashima, and H. Matsuo, "Improving apache spark's cache mechanism with lrc-based method using bloom filter," in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pp. 496–500, IEEE, 2018.

[89] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017.

[90] G. Jia, G. Han, J. Jiang, and L. Liu, "Dynamic adaptive replacement policy in shared last-level cache of dram/pcm hybrid memory for big data storage," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 4, pp. 1951–1960, 2017.

[91] T. Brecht, E. Arjomandi, C. Li, and H. Pham, "Controlling garbage collection and heap growth to reduce the execution time of java applications," in *ACM Sigplan Notices*, vol. 36, ACM, 2001.

[92] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 99–112, ACM, 2012.

[93] Y. Chen, B. Yang, A. Abraham, and L. Peng, "Automatic design of hierarchical takagi–sugeno type fuzzy systems using evolutionary algorithms," *IEEE Transactions on Fuzzy Systems*, vol. 15, no. 3, pp. 385–397, 2007.

[94] P. Lama, S. Wang, X. Zhou, and D. Cheng, "Performance isolation of data-intensive scale-out applications in a multi-tenant cloud," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 85–94, IEEE, 2018.

[95] D. Yang, W. Rang, and D. Cheng, "Joint optimization of mapreduce scheduling and network policy in hierarchical clouds," in *Proceedings of the 47th International Conference on Parallel Processing*, p. 66, ACM, 2018.

[96] D. Yang, W. Rang, D. Cheng, Y. Wang, J. Tian, and D. Tao, "Elastic executor provisioning for iterative workloads on apache spark," in *2019 IEEE International Conference on Big Data (Big Data)*, pp. 413–422, IEEE, 2019.

[97] A. Colorni, M. Dorigo, F. Maffioli, V. Maniezzo, G. Righini, and M. Trubian, "Heuristics from nature for hard combinatorial optimization problems," *International Transactions in Operational Research*, vol. 3, no. 1, pp. 1–21, 1996.

[98] H. Tian, M. Yu, and W. Wang, "Continuum: A platform for cost-aware, low-latency continual learning.," in *SoCC*, pp. 26–40, 2018.

[99] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[100] D. Lei, Y. Yuan, J. Cai, and D. Bai, "An imperialist competitive algorithm with memory for distributed unrelated parallel machines scheduling," *International Journal of Production Research*, vol. 58, no. 2, pp. 597–614, 2020.

[101] Q.-K. Pan and R. Ruiz, "An effective iterated greedy algorithm for the mixed no-idle permutation flowshop scheduling problem," *Omega*, vol. 44, pp. 41–50, 2014.

[102] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[103] X. Xia, C. Xu, and B. Nan, "Inception-v3 for flower classification," in *2017 2nd International Conference on Image, Vision and Computing (ICIVC)*, pp. 783–787, IEEE, 2017.

[104] K. Krishna and M. N. Murty, "Genetic k-means algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, no. 3, pp. 433–439, 1999.

[105] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[106] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," *arXiv preprint arXiv:2104.07857*, 2021.

[107] T. Chen, H. Liu, X. Liao, and H. Jin, "Resource abstraction and data placement for distributed hybrid memory pool," *Frontiers of Computer Science*, vol. 15, no. 3, pp. 1–11, 2021.