Cluster Computing (2019) 22:385–398
https://doi.org/10.1007/s10586-018-2844-1

A programmable shared-memory system for an array of processing-in-memory devices

Sangkuen Lee[1] · Hyogi Sim[1] · Youngjae Kim[2] · Sudharshan S. Vazhkudai[1]

Received: 3 January 2018 / Accepted: 20 August 2018 / Published online: 30 August 2018

Abstract
Processing in memory (PIM), the concept of integrating processing directly with memory has been attracting a lot of attention, since PIM can assist in overcoming the throughput limitation caused by data movement between CPU and memory. The challenge, however, is that it requires the programmers to have a deep understanding of the PIM architecture to maximize the benefits such as data locality and parallel thread execution on multiple PIM devices. In this study, we present AnalyzeThat, a programmable shared-memory system for parallel data processing with PIM devices. Thematic to AnalyzeThat is a rich PIM-aware data structure (PADS), which is an encapsulation that integrally ties together the data, the analysis tasks and the runtime needed to interface with the PIM device array. The PADS abstraction provides (i) a sophisticated key-value data container that allows programmers to easily store data on multiple PIMs, (ii) a suite of parallel operations with which users can easily implement data analysis applications, and (iii) a runtime, hidden to programmers, which provides the mechanisms needed to overlay both the data and the tasks on the PIM device array in an intelligent fashion, based on PIM-specific information collected from the hardware. We have developed a PIM emulation framework called AnalyzeThat. Our experimental evaluation with representative data analytics applications suggests that the proposed system can significantly reduce the PIM programming effort without losing its technology benefits.

Keywords Programmable devices · Storage systems · Processing-in-memory · Big dataprocessing

1 Introduction

Processing-in-memory (PIM) is a well-known concept of integrating processing units (cores) with memory devices in order to reduce memory latency and increase memory bandwidth [1]. PIM was originally introduced more than a decade ago, with several studies showing its potential advantages in various applications such as knowledge discovery, scientific computing, image processing and databases [2–6]. However, due to the difficulty in the heterogeneous manufacturing process of logic and memory, so far, PIM has not been widely adopted in commodity systems [7].

The preliminary version of the paper was published in the Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) (2017).

✉ Youngjae Kim
youkim@sogang.ac.kr

Sangkuen Lee
lees4@ornl.gov

Hyogi Sim
simh@ornl.gov

Sudharshan S. Vazhkudai
vazhkudaiss@ornl.gov

Extended author information available on the last page of the article

Recently, there has been a renewed interest in PIM from academia and industry [8]. Emergence of advanced 3D memory allows the stacking of memory chips atop a processing unit (e.g., GPU), enabling processing near memory, e.g., TOP-PIM [9] that was put forward by AMD. Even without 3D memory, specialized PIM devices have been prototyped by Micron in their "Automata Processor" in November 2013, which is a DRAM chip with an array of processors [10]. NVRAM such as PRAM [11], memristors [12], and STT-MRAM [13] is expected to replace DRAM as it is non-volatile and power-efficient. PIM architectures can also employ such emerging memory in place of DRAM. However, since NVRAM is limited in write cycles, the write frequency on the memory needs to be carefully controlled.

Exploiting PIM architectures has potential advantages for processing big data in terms of both energy efficiency and processing time [7, 14]. Not only can multiple PIM cores process data in a parallel fashion, but each PIM core can also achieve higher data processing throughput than commodity systems by accessing data stored in its corresponding local memory device with lower latency [9, 14, 15]. The concept of PIM is now also acknowledged to be useful in extreme-scale systems, where power consumption is increasingly becoming a significant design constraint.

For example, the U.S. Department of Energy's (DOE) CORAL consortium is deploying three O(100) petaflops systems, SUMMIT, SEIRRA and AURORA systems at Oak Ridge National Laboratory (ORNL), Lawrence Livermore National Laboratory (LLNL) and Argonne National Laboratory (ANL), respectively in the 2018-2019 timeframe, which are expected to consume in the range of 10-13MW of power. These systems will be equipped with deep memory tiers ranging from tens of GBs of high-bandwidth memory (HBM), several PBs of DRAM and persistent memory. In such systems, DRAM is a significant source of power consumption. The DOE's future exascale system in 2023 is expected to be built within an energy envelope of 20MW. At exascale, it is widely expected that the cost of data movement between the deep memory tiers will rival the cost of computation itself [16]. While technologies such as HBM and persistent memory help alleviate this concern, PIM and processing near memory can be a significant step in this direction as well.

However, it is a significant challenge to integrate PIM architectures into extant user application software. Programming the PIM architecture can be a non-intuitive task for users, since it is necessary to properly distribute data and tasks to multiple PIM devices to fully take advantage of data locality and parallelism of the PIM devices [9]. If memory allocation and concurrency control are not accomplished properly, efficiency and scalability of the application can significantly decrease, due to data skew [17] and massive remote access [18].

To address these challenges, we present AnalyzeThat, a programmable shared memory system for PIM devices [19]. The system provides an abstraction for parallel data processing with PIM devices, which allows programmers to focus on the functionality of their programs and not on the management of data placement and thread concurrency. More specifically, the abstraction is provided as a PIM-aware data structure (PADS), which is a collection of key-value pairs distributed over multiple PIM devices. Programmers load their data into the PADS objects and execute the workflow of data analytics applications with PADS parallel operations. Thereafter, AnalyzeThat's runtime is responsible for making decisions to efficiently process the PADS parallel operations on a system having an array of PIM devices. The decisions (e.g., how to distribute key-value pairs and which PIM to offload a task to) can be made based on PIM-specific information that is collected from hardware devices. For such a capability, AnalyzeThat exploits operating system and hardware support (e.g., device driver and global address space). In addition, in order to give more control to programmers to achieve generality of PIM programming, a low-level PIM programming library that resembles POSIX pthread APIs is provided as part of the AnalyzeThat library suite.

Contributions The contributions of AnalyzeThat are as follows. First, we identify the support required from the operating system and the hardware. Based on that, we present fundamental kernel and user-level software that are necessary for providing PIM programming capability to users, including a PIM device driver and a low-level PIM programming library. Second, we present a PIM-aware data structure (PADS) and PADS operations to lower the entry barrier for programming with PIM-augmented architectures, without losing the benefits of such an architecture. Further, we present techniques for minimizing the overhead of the executing operations. Third, we show how AnalyzeThat can be exploited for various data analysis applications such as statistics, text-processing and graph-processing, via an emulation platform. Our experimental results confirm that a wide range of data analysis applications, implemented using AnalyzeThat's PADS, can provide better performance compared to using a host processor.

2 Related work

Recent studies have identified that a PIM architecture suits large-scale data analysis workloads due to its parallelism and fast memory access [7, 14, 20]. However, despite the significant potential, the weak support for a high-level

programming interface brings new challenges. One solution is to extend the existing NUMA (non-uniform memory access) library to manage the PIM devices and allow programmers to precisely control each one of them, e.g., allocating space and executing a kernel on a PIM device. However, it requires a deep understanding of hardware-specific details and may hinder programmers from focusing on the application logic itself. OpenCL [21], which provides a generic interface for various accelerators, currently does not support the PIM architecture. More importantly, the OpenCL API still requires programmers to understand the architecture of the accelerator hardware.

MapReduce [22] has been widely adopted to write parallel data analysis applications, not only on commodity cluster systems [23] but also on shared-memory machines [24] and GPU augmented systems [25]. Spark [26] provides a powerful distributed data abstraction, RDD, which exploits memory space across cluster nodes and provides a similar interface to MapReduce. Several studies have shown the potential of performing MapReduce workloads on PIM architectures [7, 27]. However, these studies primarily address the performance impact of the PIM architecture, but not the software architecture and usability, e.g., the runtime system to orchestrate the data placement and code execution.

3 AnalyzeThat programmable system

Here, we discuss our key design principles, and provide the architectural overview of the AnalyzeThat system.

3.1 Goals

Easy programming Interface Our main objective is to provide an easy and effective programming interface for PIM-augmented systems, so that users can easily program the system without having to understand system-specific details such as memory management, thread operation, and non-uniform memory access latency.

Reduce data movement The location of the data, i.e., which PIM device it resides on, determines the data movement cost, and it is expected that the data movement cost will compete with computation cost. The reason being, the analysis kernel running on a particular PIM core needs to fetch the data from the remote PIM device if it is not on its local memory. Thus, our system aims to minimize the data movement cost between the host and the PIM devices, and across the PIM devices during the application execution. In addition, it is necessary to optimize data placement by factoring in PIM-specific information, such as the computing load and memory usage, which will affect performance.

Programming flexibility and generality Hiding details such as how the PIM devices are used from the programmers can improve programmability. However, it is also necessary to give advanced programmers more controls (e.g., manually place a piece of data object or offload a task to a specific PIM device). Thus, we also aim to provide a set of low-level PIM programming library for this purpose.

3.2 Overview

We envision AnalyzeThat as a programmable shared memory system atop an array of PIM devices, in order to process data in-situ, on the memory device where they already reside. We argue that such an approach helps minimize data movement costs on future systems. Figure 1 depicts the interactions between various components of AnalyzeThat.

Array of PIM devices At the lowest level is an array of PIM devices, capable of processing. As we will discuss in Sect. 4.1, emerging hardware technologies support a single shared memory address space for a node composed of general CPUs and compute accelerators such as PIM, where any core can globally access any memory region [28, 29]. The PIM device can either be 3D stacked with memory chips layered atop the logic chip on the same die or a discreet PIM device with an embedded controller.

PIM device driver We have developed a first order implementation of a device driver that each PIM device needs to support in order to realize the functionality needed in the AnalyzeThat system. In our implementation, the device driver is responsible for providing the communication path between the PIM hardware and the higher-level components in AnalyzeThat, maintaining PIM-specific internal information (e.g., wearout, data load) that will be queried for data placement, task offloading and data segment expansion.

Low-level PIM programming library Atop the PIM array and the device driver is a low-level PIM programming library,

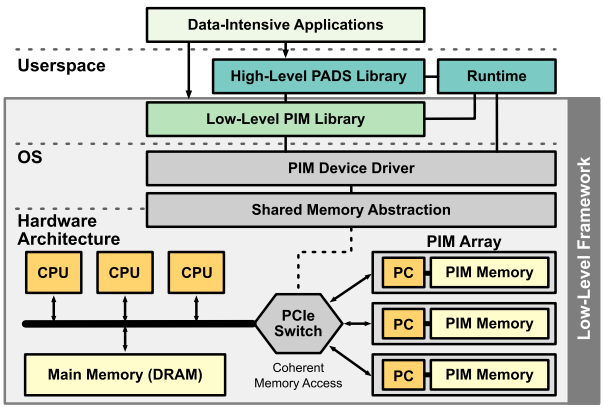


Fig. 1 The hardware and software architecture of AnalyzeThat. PC denotes the PIM-core

which gives direct control of PIM devices to programmers, such as allocating PIM memory and offloading a task to a PIM device. An advanced programmer can implement his application directly using the low-level library.

PADS (PIM-aware data structure) The PADS library provides an easier way of programming with PIM devices. PADS provides an object-oriented abstraction that tightly couples a data object and its operations. Specifically, PADS provides an abstraction of a *key-value container* and hides hardware-specific details (e.g., a PIM location) from programmers. In addition, the PADS key-value container supports a *suite of parallel operations*, such as *map()* and *reduce()*, which can be used to manipulate the key-value dataset in parallel. For instance, a programmer can create a PADS object and populate it from an input file. The PADS library then transparently distributes the input data across multiple PIM devices based on a data placement policy. Thereafter, the programmer can perform data manipulation in parallel by invoking the PADS operations.

Runtime environment Behind the PADS abstraction, a runtime handles the hardware-specific details such as data distribution and task execution across the PIM array. For this, the runtime periodically collects the status of each PIM device using the low-level library and the device driver. Based on the device status, the runtime dynamically makes decisions on data and task load distribution (Table 1).

AnalyzeThat utilizes the PIM cores to process the data in parallel. Together, these constructs provide a very potent, programmable shared memory abstraction for in-situ data analytics atop an array of PIM devices.

4 AnalyzeThat low-level framework

The low-level framework (Fig. 1) within AnalyzeThat consists of an array of PIM devices, operating system including the PIM device drivers and the low-level PIM userspace APIs. In particular, the high-level programming interface of AnalyzeThat (PADS, Sect. 5) is built upon a unified shared memory abstraction across heterogeneous memory devices, i.e., main memory and PIM memories that this low-level framework provides.

4.1 Hardware architecture

Each PIM device in AnalyzeThat is composed of a dedicated computing unit (PIM core), a set of programmable registers and memory chips (PIM memory). The PIM core is a *fully programmable* low-power processor similar to ARM processors [30, 31]. This allows any general program to be executed on the devices and, therefore, grants more flexible programming than FPGA-based accelerators [32]. However, due to the difference in the instruction set architectures between the host CPU and the PIM core, the target binary code that is to be executed on the PIM core should be compiled and built separately with a supporting compiler. In addition to general programmability, each PIM core contains its own hardware cache and MMU (Memory Management Unit), and runs a firmware to control the internal hardware operations. The programmable registers can be read and written by host applications to initiate a task execution or to fetch runtime information, e.g., memory duty cycles. Such information is used by the AnalyzeThat runtime (Sect. 6).

As shown in Fig. 1, multiple PIM devices are connected to a single host via a fast switch interconnect, i.e., PCI Express. AnalyzeThat exploits the emerging memory interconnect protocol, i.e., Cache Coherent Interconnect for Accelerators (CCIX) [28], which allows cache coherent accesses across heterogeneous memory devices from different processors and accelerators. Note that CCIX protocol does not require any modifications to existing system software or operating systems, because the protocol is fully implemented in hardware and firmware [28]. The coherent memory access protocol and the fast switch interconnect allow host CPUs to directly access the PIM memories

Table 1 Example APIs of low-level PIM library in C and high-level PADS library in C++

Low-level	`void* pimmalloc(size_t size, int pim)`	Allocates `size` bytes of memory in PIM device `pim`
	`void pimfree(void* addr)`	Frees memory of address `addr`
	`int pimexec_exec(pimexec_data_t* pe)`	Initiate offloading of a user-defined function
	`int pimexec_wait(pimexec_data_t* pe)`	Block the current thread until the execution completes
PADS	`void PADS.import(char* file, parser_t* pf)`	Import data from a `file` using a parser function `pf`
	`void PADS.map(PADS& out, mapper_t* mf, void* arg)`	Performs a user-defined map function `mf` and stores results in `out`
	`void PADS.reduce(PADS& out, reducer_t* rf, void* arg)`	Performs a user-defined reduce function `rf` and stores results in `out`
	`void PADS.export(char* file)`	Export data into `file`

`pimexec_data_t` is a record type that encapsulates all information regarding a code execution on a PIM core

without having to explicitly transfer data between the host and the PIM memories. Similarly, a PIM core can access not only its own local PIM memory but also other remote PIM memories and DRAM on the host. Overall, this shared memory abstraction from the hardware enables us to project a consistent virtual address space to both the offloaded PIM task and its parent application on the host, e.g., a memory pointer can be shared between them, and greatly facilitates application development.

4.2 PIM device driver

In addition to its capability as a memory storage device, the PIM device features its own processing power. To exploit the processing power, e.g., execute a task using a PIM core, a host software needs to communicate with a PIM device, which requires access to the programmable registers of the PIM device. The PIM device driver primarily assists userspace programs to allocate memory space and also to access the programmable registers by providing a memory-mapped I/O interface. Specifically, during system initialization, the device driver detects available PIM devices that are connected to the host. For each PIM device, the device driver then creates a dedicated device file, i.e., `/dev/pimN`, and */proc* entries, i.e., `/proc/pim/pimN/` on the host. Also, a set of *ioctl()* operations through the device files are supported for userspace applications. For instance, to allocate memory space on a specific PIM device (e.g., 40 KB memory allocation on the first PIM device), an application first opens the device file (e.g., `/dev/pim0`) and invokes the *ioctl()* system call with a predefined operation id (e.g., `PIM_IOC_GETPAGES`) and the amount of requesting space in the number of pages (e.g., 10). Similarly, to initiate a task execution on a PIM device, the application invokes *ioctl()* with a pointer to a structure containing target function and argument addresses, and a dedicated operation id (e.g., `PIM_IOC_EXECUTE`). In addition, the device driver supports *ioctl()* operations that allow host applications to access PIM-internal runtime information such as PIM core utilization and memory usage. In our design, the physical space allocation of PIM memory is managed by device driver. Current allocation status of each PIM memory can be acquired by reading the */proc* entry.

4.3 Low-level PIM library

The low-level PIM library of AnalyzeThat is layered atop the PIM device driver and consists of a set of functions that allow users to manually control PIM devices. Its usage is similar to that of POSIX dynamic memory and pthread functions. The library primarily provides two functionalities—memory management and task offloading, by wrapping the low-level *ioctl* interface. For memory management, it provides *pimmalloc()* and *pimfree()*. Programmers can allocate memory using the *pimmalloc()* function similar to the standard *malloc()* function, with an additional argument of *pim_id* to specify a PIM device that the space is allocated from. Similar to the *malloc()* function that internally invokes the *brk()* system call to extend the data segment if needed, the *pimmalloc()* requests the device driver via the aforementioned *ioctl()* interface to allocate memory pages and expand the data segment. Note that the device driver globally synchronizes memory allocation requests from multiple applications, and grants applications access to acquire a particular memory region. To offload tasks to the PIM cores, the library provides *pimexec_exec()*. Programmers can offload a function to a specific PIM core by providing the pointer of the function and a *pim_id*. For functions running in parallel on PIM cores, programmers can synchronize the tasks using the *pimexec_wait()*, which forces a wait for the specified thread to terminate.

Although the low-level library could have been implemented with an existing heterogeneous computing framework (i.e., OpenCL [21]), we adopt a PIM-specific framework since OpenCL is known to sacrifice the performance to provide portability among different hardwares [33].

5 PADS: AnalyzeThat programming interface

While the low-level library (Sect. 4.3) provides direct access to the PIM devices to advanced programmers, the PIM-aware data structure (PADS) is a higher-level data abstraction that hides the intricate details of the hardware. Thematic to PADS is the encapsulation of data, the analysis to be performed on the data and the mechanisms to overlay both the data and the analysis on the PIM device array. Users only need to create and manipulate the data structure in order to take advantage of the PIM functionality, while remaining oblivious to the complexity of the hardware. To this end, the PADS abstraction is composed of two components, namely the *key-value container* and *parallel operations*.

5.1 PADS key-value container

We have chosen to represent the data within PADS using a key-value container, i.e., data is stored as key-value pairs. Key-value pair representation of data is simple yet generic enough to be used for various data analysis applications. Various data analysis systems and modern databases adopt key-value pair representation [22, 34, 35]. Internally, a PADS key-value container consists of n sub-containers, each of which is associated with a single PIM device. To

use the PADS data structure, an application first creates a PADS key-value container object (referred to as a PADS object, hereafter). Then, the application can simply put key-value pairs into the PADS object, similar to using other familiar data containers, e.g., C++ *queue*, *set*, *map*, etc. The application does not have to specify the sub-container that internally stores the key-value pair. Instead, the AnalyzeThat runtime transparently selects a sub-container based on a data placement policy, as we will explain further in Sect. 6.2.

5.2 PADS operations

PADS supports a set of operations that run on the associated PADS object. This includes operations that can facilitate data analysis tasks on the PADS object. Many data analysis tasks in practice take input data from files. Manually parsing a file and converting raw data to a structured record set is a tedious, error-prone task. PADS provides an *import()* function that automatically parses a given input file and populates a PADS object with the parsed key-value pairs. It supports many popular formats such as txt, csv and netCDF, and a user can also specify his own parser function. Similarly, the *export()* function writes all key-value pairs in a PADS object to a file in a specified format. In addition, PADS allows users to perform customized data processing via the popular *map()* and *reduce()* interface [22]. The *map()* function, which is a member function of the PADS object, takes a user-defined function and an output PADS object as arguments. The user-defined function is executed on every key-value pair in the PADS object. The *reduce()* function groups key-value pairs in a PADS object based on their keys and produces a single, reduced key-value pair for each key group.

Figure 2 illustrates an example program that uses the PADS library. We assume that input data is stored in a file (*input.csv*), and each record in the file is a pair of a string key and an integer value, i.e, $<key, value>$. The program is to double (i.e., $\times$ 2) each number and, for each distinct key, sum all the numbers sharing the same key. First, the program creates three PADS objects, *input* for input data, *mapped* for intermediate data, and *reduced* for final result data (line 12). To populate key-value pairs from the input file, it simply calls the *import()* function which automatically parses the csv file and populates *input* (line 14). Next, it invokes the *map()* function with the user-defined function *timesTwo()* (line 15). During this *map()* processing, the *timesTwo()* function, which doubles and updates a given value (line 2), is internally called for every key-value pair in the PADS object, *input*. To calculate the sum of numbers grouped by keys (line 16), the program calls *reduce()* with the custom reduce function, *sumByKey()*. The final result is stored to a file by the *export()* function (line 17). Note that

```
1  // user-defined map function
2  void timesTwo(char* k, char* v, PADS& pd, void *arg) {
3      pd.insert(k, stoi(v) * 2);
4  }
5  // user-define reduce function
6  void sumByKey(char *k, char *v, char *out, void *arg) {
7      sprintf(out, "%d", stoi(out) + stoi(v));
8  }
9  // main program
10 int main(void) {
11     PADS input, mapped, reduced;
12
13     input.import("input.csv");
14     input.map(mapped, timesTwo, NULL);
15     mapped.reduce(reduced, sumByKey, NULL);
16     reduced.export("result.csv");
17 }
```

Fig. 2 Example of using PADS library in a C++ program

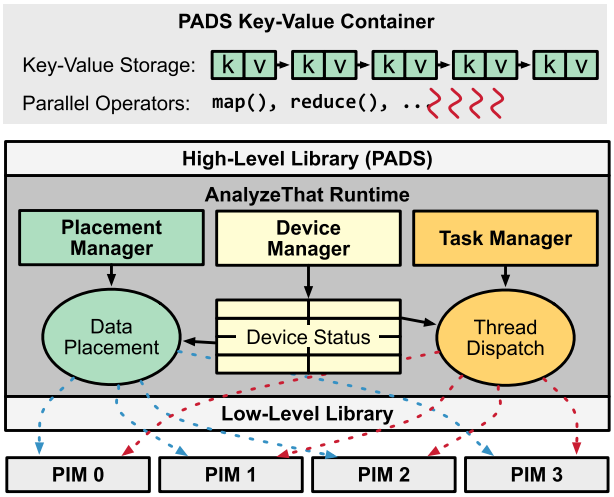


Fig. 3 Users can write applications with the PADS library, and AnalyzeThat runtime transparently manages PIM-specific details

the PADS abstraction hides the hardware-specific details, and a programmer does not need to perform manual optimizations regarding the underlying PIM architecture.

6 AnalyzeThat runtime

When programming with the high-level PADS abstraction (Sect. 5), users can focus on writing application logic itself without having to understand the PIM hardware architecture. Below the PADS abstraction, the AnalyzeThat runtime transparently handles PIM hardware-specific details, i.e., data distribution and parallel task execution across the PIM array. Figure 3 shows the internal architecture of the runtime. Users can simply populate a PADS key-value container and perform parallel operations, e.g., *map()* and *reduce()*. The runtime consists of the device manager, placement manager and task manager, and internally handles all PADS operations with regard to the underlying PIM hardware architecture.

6.1 Device manager

To optimize the performance, the runtime should consistently keep track of the device status. For instance, the runtime should know which PIM core is busy before assigning tasks to the PIM device. Likewise, to evenly distribute data load, the runtime needs to track which PIM memory is most- or least-populated. To this end, the Device Manager in the runtime periodically gathers PIM device-specific information and stores each device's utilization statistics into a global device status table in memory. The records in the device status table are updated every 3 s, which is tunable. In particular, device-specific information, e.g., memory wearout status and PIM core utilization, is directly fetched from the device via *ioctl()* system call (Sect. 4.3). The device utilization statistics can be gathered directly from the PADS objects. The device status table is referenced by the Placement Manager and the Task Manager for making runtime decisions such as data placement and PIM thread dispatch, respectively.

6.2 Placement manager

When an application inserts a key-value pair into a PADS container (e.g., *pd.insert()* in Fig. 2), the Placement Manager selects a sub-container that is associated with a single PIM device. Particularly, the decision is made based on the following three objectives: (1) Balancing the load evenly across the PIM array. Load imbalance can slow down the overall execution time, which is determined by the completion time of the slowest PIM task. (2) Grouping key-value pairs in a single PIM device based on their keys. The *reduce()* operation runs faster if the key-value pairs are already grouped by their keys. In addition, the absence of proper aggregation may incur frequent remote PIM accesses, which can lead to a significant performance drop. (3) Storing output key-value pairs in a local PIM memory. Again, accessing local PIM memory is substantially faster than accessing remote PIM memory. Moreover, it eliminates potential lock contention among multiple PIM threads. Based on these objectives, the Placement Manager implements the following three *static* data placement policies.

Round-Robin (RR) In RR, the runtime selects a PIM device for storing data in a circular, round-robin order. Each PIM device p_i maintains a counter c_i, which is initialized to i. When a PIM thread running on p_i inserts a key-value pair, the runtime stores the key-value pair in $p_{mod(c_i,n)}$ and increases the counter c_i by 1. While RR evenly distributes data across the PIM devices, it does not aggregate key-value pairs for reducing the remote access.

Local-assignment (LA) LA always places a key-value pair to the same PIM device p_i that a requesting PIM thread is running on. This can maximize the local PIM memory accesses but does not guarantee an even data distribution.

Hashing (HS) HS uses a hash function $h(k) \rightarrow [0, n-1]$, where n is the number of PIM devices, to select a PIM device to store a key-value pair $<k, v>$. This effectively aggregates key-value pairs based on the key, but a skewed distribution of the data keys may directly lead to a load imbalance across the PIM array.

Algorithm 1: Dynamic (DY) data placement policy.

```
   Input: (k, v) // key-value pair to insert
   Output: p_i // PIM device ID
 1 if check_avoid(hashing(k))==false then
 2 |   return hashing(k);
 3 else
 4 |   if check_avoid(p_local)==false then
 5 |   |   return p_local; // Local PIM ID
 6 |   else
 7 |   |   if check_avoid(p_min)==false then
 8 |   |   |   return p_min; // PIM ID with the min. key-value pairs
 9 |   |   else
10 |   |   |   return p_min_wear; // PIM with the min. wearout level
11 |   |   end
12 |   end
13 end
14 // definition of check_avoid()
   Input: p_i // PIM device ID
   Output: true or false
15 if p_i==p_max then
16 |   return true;
17 else
18 |   with probability (1-w),
19 |   goto skip_wearout_check;
20 |   if p_i==p_max_wear then
21 |   |   return true;
22 |   end
23 |   skip_wearout_check:
24 |   return false;
25 end
```

Although these three static policies can optimize an individual data placement objective, they cannot adapt to the workload changes in practice. Therefore, the Placement Manager provides an additional data placement policy, Dynamic, which *dynamically* balances the three objectives.

Dynamic (DY) DY is a hybrid approach that merges the other three policies, and tries to avoid the load imbalance and skewed memory utilization based on the runtime status. Specifically, DY first works identical to one of three data placement policies, i.e., RR, LA or HS. However, if a decision of the chosen data placement policy violates predefined conditions, DY tries the next data placement policy to select a target PIM device. This algorithm is shown in Algorithm 1. The *check_avoid()* function determines whether a given PIM ID violates any predefined conditions, i.e., it returns *true* if the given PIM device contains the most number of key-value pairs. In addition, for NVRAM-based PIM devices, DY can distribute data in a wearout-aware fashion to prevent early device retirements. In particular, based on the value of the tunable parameter w ($0 \leq w \leq 1$), DY avoids the data placement to the PIM device with the high wearout-level. For example, with $w = 0.5$, the *check_avoid()* function factors the device wearout when determining the target PIM locations for the half (50%) of the data.

6.3 Task manager

Task Manager is responsible for the execution of the PADS application program by utilizing the PIM-architecture, i.e., fast local memory access and parallelism. Specifically, to maximize the local PIM memory access, the Task Manager spawns a PIM thread on every PIM device. Each PIM thread then executes the user-defined PADS operation, which is tightly associated with the PADS key-value pairs in the local PIM memory. Internally, the runtime exploits the low-level PIM library functions (Sect. 4.3) to accomplish task offloading. In addition, the Task Manager also optimizes the PADS operations based on workload characteristics. For instance, the *reduce()* operation gathers results from all PIM devices and stores into a single PIM device. For certain workloads, such a global *reduce()* operation 5.2 can incur a significant contention in the destination PIM device, where result data are collected. To mitigate this contention, the Task Manager can perform a Local Reduce (LR), in which each PIM thread performs the *reduce()* operation locally by creating an intermediate PADS object in the local PIM memory. This effectively eliminates remote PIM memory accesses. After the Local Reduce, the global *reduce()* takes the sorted key-value pairs in the intermediate PADS objects and creates the final result. As we will see later (Sect. 7), the Local Reduce can significantly reduce the concurrent remote PIM memory accesses, particularly when many key-value pairs across the PIM array share the same keys.

7 Evaluation

Implementation We have implemented an emulation framework for AnalyzeThat using 5000 lines of C/C++ code. We emulated the PIM device by preallocating the system memory and binding threads to cores. To emulate the different access characteristics of the PIM memory, we introduced delays according to the memory access types, i.e., access from the host core or the PIM core. Specifically, a thread stays in a busy loop until the processor timestamp counter (TSC) reaches a desired value. The delay is computed based on a relative delay from a baseline when the PIM core accesses its local memory. When the PIM core accesses remote PIM memory, we add a delay corresponding to the time taken to access remote memory in a NUMA node. Our measurements of local and remote memory bandwidth on a NUMA node are 6733.7 MB/s and 4567.5 MB/s, respectively, i.e., remote memory is 32.2% slower than local memory. When the PIM accesses the remote memory, either the DRAM on the host or memory on another PIM, we add a 32% delay. Also, in our setup the DRAM access is 4× slower than PIM's local memory access [9].

Testbed Our test machine comprised of two processors (1.8 GHz Intel Xeon E5-2603, each with four cores), 64 GB RAM and ran the RedHat Enterprise Linux 6.5 with the 3.1.22 kernel. We dedicated one core for the host-side processing and emulates up to seven PIM devices with the remaining seven cores. For each emulated PIM device, we preallocated 4 GB of host memory and decreased the clock speed of the core to 1.2 GHz.

7.1 Programmability of PADS

First, we demonstrate the effectiveness of the PADS library. Using the PADS library, we implemented five representative big data applications in a few tens of lines of code, e.g., 65 lines at most for PageRank. Here, we briefly explain each application and our implementation.

Group By Aggregation (GAG) computes the statistical summary, e.g., the total sum and average value from numerical datasets. Each line of the input file is composed of key-value pairs, delimited by a comma (","). The program first parses the input file and produces a set of key-value pairs. It then performs calculations by grouping the key-value pairs on the same key. Figure 4a shows the code snippet of our implementation. In the code snippet, *aggMap()* function appends "1" to the value of each key-value pair in *data*, and inserts the key-value pair into *mapped*.

```
// (a) Group-by-Aggregation and Aggregation
void aggMap(char *k, char *v, PADS& t) {
    strcpy(new_val, v);
    strcat(new_val, "1");
    t.insert(k, new_val); // "A" instead of k for AG
}
char *aggReduce(char *k, char *v, char *reduced) {
    tokenizer(v, head, tail, "␣");
    tokenizer(reduced, sum, int, avg, "␣");
    sum = stoi(sum) + stoi(head)
    cnt = stoi(cnt) + stoi(int)
    avg = (double) sum / cnt;
    sprintf(reduced, "%d␣%d␣%f", sum, int, avg);
}
int main(void) {
    PADS data, mapped, result;
    data.import("input.txt");
    data.map(mapped, aggMap);
    mapped.reduce(result, aggReduce);
}
// (b) Grep
void grepMap(char *k, char *v, char *arg, PADS& t) {
    if (strstr(v, arg))
        t.insert(k, v);
}
int main(void) {
    PADS data, result;
    data.import("input.txt");
    data.map(result, grepMap, "bob");
}
// (c) Word-Count
void wordCountMap(char *k, char *v, PADS& t) {
    while ((token = strsep(&v, "␣")) != NULL)
        t.insert(k, 1);
}
int main(void) {
    PADS data, mapped, result;
    data.import("input.txt");
    data.map(mapped, wordCountMap);
    mapped.reduce(result, sumByKey);
}
// (d) Page-Rank
void prMap(char *k, char *v, PADS& t) {
    // distribute the score to adjacent nodes
}
void prReduce(char *k, char *v, char *reduced) {
    // aggregate scores from adjacent nodes
}
int main(void) {
    pr_init[0].import("graph.txt");
    for (i = 0; i < iter_no; i++) {
        pr_init[i].map(pr_map[i], prMap);
        pr_map[i].reduce(pr_reduce[i], prReduce);
        pr_reduce[i].map(pr_adjust[i], prMapAdjust);
        pr_init[i+1] = pr_adjust[i];
    }
}
```

Fig. 4 Implementation overview of data analysis applications using PADS library

Then, *aggReduce()* performs calculations by aggregating all values in *mapped* based on their keys and stores the result in *result*.

Aggregation (AG) works similar to GAG but calculates the global statistical summary, i.e., not based on keys. We implemented AG with a slight modification to the GAG program. In particular, we replace key in every key-value pair with the same value (e.g., 'A') in *aggMap()*, to make all key-value pairs share the same key. Therefore, *aggReduce()* calculates the statistical summary of all key-value pairs.

Grep (GR) is a string matching application that prints lines containing a matching keyword from the input file. In our implementation (Fig. 4b), each line of the input file is parsed as a $<line\_no, text>$. *grMap()* passes a key-value pair to the output *target*, only if the *text* contains the keyword. Note that the reduce operation is not used here.

WordCount (WC) counts the occurrences of each word in the input text file. Each line is parsed as a $<line\_no, text>$ pair. *wcMap()* then generates $<word, 1>$ for each pair, which is appended to *target*. *sumByKey()* in Figure 2 is used to aggregate the result.

PageRank (PR) computes the ranking scores of all nodes in a graph dataset [36]. PR is an iterative algorithm and more complex than the other workloads, as it requires four user-defined functions (one parser, two map, and one reduce functions). In Fig. 4d, we assume that each line of the input file describes the edges between the nodes in a graph, e.g., a line "2, 3 4" describes that the node "2" is connected to the nodes "3" and "4" in the graph. *graphParser()* loads the input data and assigns an initial PageRank score (i.e.,"1") to every node, which is appended to the original value with a delimiter (e.g., key='2', value='3 4 ||1'). Then, PageRank scores are iteratively updated as follows. First, *prMap()* equally distributes every node's PageRank score to its adjacent nodes. Second, for each node, *prReduce()* computes a new PageRanks score for the node by aggregating the scores distributed from adjacent nodes. Next, the score of each node is adjusted by the *prMapAdjust()* function. After the ith iteration, each key-value pair in $pr\_init[i+1]$ represents its adjacent nodes and PageRank score of a given node (e.g., key='2', value='3 4 ||0.125') at the iteration.

For the rest of this section, we use these five applications to study the performance of AnalyzeThat. In particular, for AG and GAG, we synthetically generated the input data consisting of 100 million entries (850 MB) with a normal distribution. For GR and WC, we used a 6.4 GB text file that concatenates contents of a wiki site [37]. For PR, a DBLP dataset (8.6 MB) [38] with 317,080 nodes and 1,049,866 edges was used. Each experiment was repeated five times, and we report the average with the 95% confidence interval.

7.2 AnalyzeThat performance

Here, we compared the AnalyzeThat performance of running applications from Sect. 7.1 against a host-based approach. For AnalyzeThat, we also evaluated four different data placement policies, i.e., Round-Robin (RR), Hashing (HS), Local-Assignment (LA) and Dynamic (DY) (Sect. 5).

Figure 5 shows the application runtimes of AnalyzeThat and the host-based approach. The results are normalized to

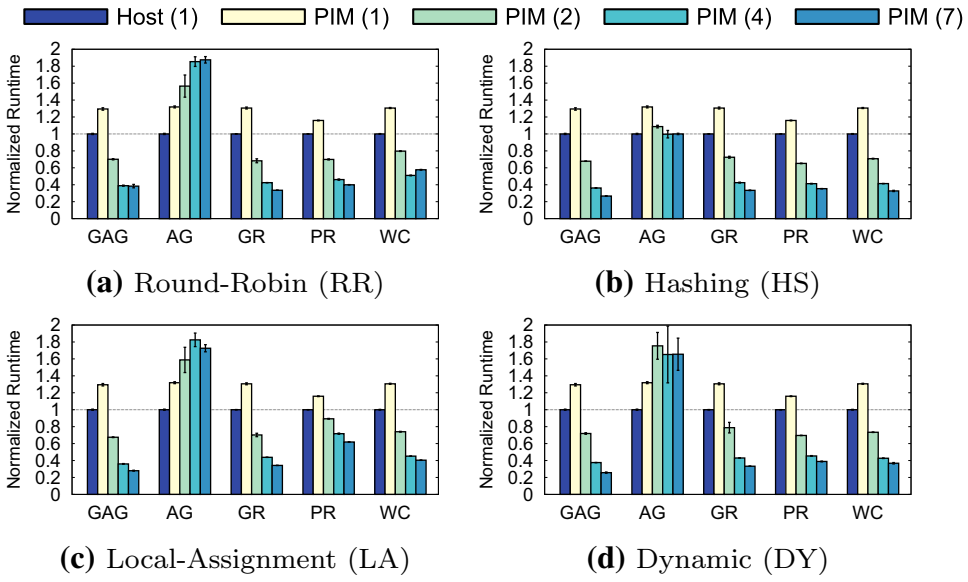


Fig. 5 Comparison of AnalyzeThat for different data placement policies, *without* Local Reduce. DY does not run wearout-aware algorithm ($w = 0$). Host(1) refers to the host with a single CPU, and PIM(n) refers to n PIM devices connected to a single host. Runtimes of Host(1) for GAG, AG, GR, and PR were 405.7, 356.8, 189.8, 44.1, and 1089.0 s, respectively

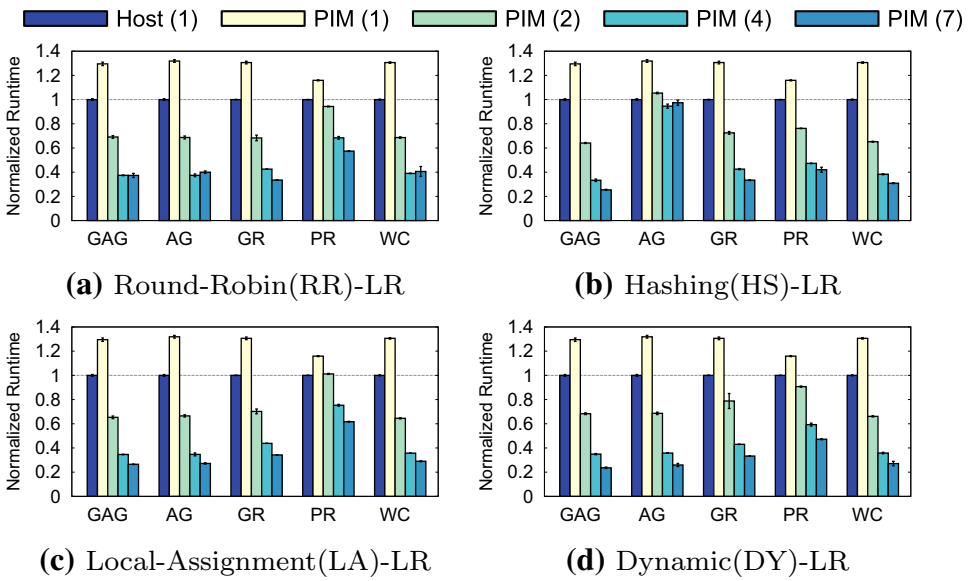


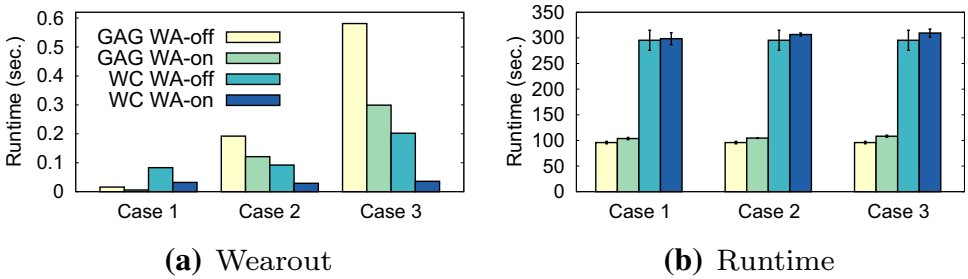
Fig. 6 Performance impact of Local-Reduce (LR) with different data placement policies. DY does not run the wearout-aware algorithm ($w = 0$). Runtimes of Host(1) for GAG, AG, GR, and PR were 405.7, 356.8, 189.8, 44.1, and 1089.0 s, respectively

the runtimes of the host-based approach. We observe that AnalyzeThat with a single PIM (PIM(1)) is 15–30% slower than the host-based approach with a single core (Host(1)). Even though the PIM internal memory bandwidth is greater than the host DRAM bandwidth, most workloads are compute-intensive, and thus the more powerful host CPU outweighs the higher memory bandwidth. However, AnalyzeThat with two PIM devices (PIM(2)) outperforms the host-based approach by 20–30%. In addition, AnalyzeThat scales almost linearly as we use more PIM devices, except for AG. For AG, the runtimes increase as more PIMs are used due to its workload characteristics (Fig. 5a, c, d). In addition, HS shows a different trend from the other policies for the AG workload. This is because HS places all intermediate data with the same key into a single PIM device.

Table 2 The test cases with different initial wearout distributions across seven PIM devices

	Case 1	Case 2	Case 3
Total (MB)	1000.00	1000.00	1000.00
Maximum (MB)	142.86	250.00	666.67
Minimum (MB)	142.86	35.71	0.00
Standard Deviation (MB)	0.00	77.15	243.98

Although this can eliminate a potential lock contention, all reduce tasks are serialized regardless of the number of available PIM devices. For GR, we observe little performance variance across the different data placement



(a) Wearout **(b)** Runtime

Fig. 7 Effects of enabling wearout-awareness of the DY data placement policy in three test cases (Table 2). We used seven PIMs and enabled Local-Reduce (LR) for all tests

policies. This is because GR is implemented only with *map()* operations (Sect. 7.1) and not affected by the performance variance of the *reduce()* operation, i.e., the lock contention and skewed distribution of the intermediate data.

7.3 Effects of enabling local-reduce technique

Next, we study the performance impact of the Local-Reduce (LR) technique, which aims to mitigate the overhead from the lock contention during the *reduce()* operation. Note that the previous experiment was conducted without employing Local-Reduce. We run the same experiments of the preceding section (Sect. 7.2), but with enabling Local-Reduce. Figure 6 shows the result.

First, we observe significant performance improvement in AG, for all data placement policies except HS. Local-Reduce effectively mitigate the lock contention during the *reduce()* operation by locally aggregating intermediate data on each PIM device. However, AG with HS does not benefit from Local-Reduce, because intermediate data is already placed in a single PIM device without any potential lock contention. Although Local-Reduce noticeably improves the performance in most workloads (AG, GAG and WC), it increases the runtime by 10–15% in PR, regardless of the data placement policy. Note that Local-Reduce is beneficial only when intermediate key-value pairs from the *map()* operation are evenly distributed across the PIM devices. However, intermediate data of PR are rather skewed because the initial degree distribution of the input graph follows the power-law distribution [39], similarly to many real-world graphs. Therefore, performing *reduce()* locally does not effectively reduce the size of the intermediate data but introduces non-negligible overhead from performing additional operations. We also observe that Dynamic with Local-Reduce (DY-LR) consistently performs better than RR-LR for all applications. In contrast, HS-LR and LA-LR performs worse than RR-LR for workloads AG and PR, respectively. Note that DY-LR performs the best in all applications except PR, for which HS-LR outperforms DY-LR by 11% with seven PIM devices (PIM(7)). However, compared to DY-LR, HS-LR does not performs steadily over different workload characteristics, e.g., HS-LR does not scale with the AG workload. Therefore, we argue that DY-LR performs best overall.

7.4 Impact of wearout-aware placement algorithm

Next, we consider a PIM device with non-volatile memory, which can have wearout issues, and study the effects of wearout-aware data placement policy. Note that only DY is wearout-aware among our four data placement policies (Sect. 6.2). To validate the effectiveness of the wearout-aware DY, we consider three different cases based on the initial wearout distribution across the PIM array, as shown in Table 2. Each case represents a different wearout status of the PIM array after 1 GB of data has been written. For instance, the data was evenly written across the PIM devices in Case 1. In Case 2 and Case 3, 250 and 666.67 MB data was written to the most populated PIM device, respectively. In Case 3, one PIM device was not utilized at all during the population. We ran GAG and WC workloads with DY-LR (Dynamic with Local-Reduce) data placement with and without the wearout-awareness. Particularly, we study how application runtime and device lifetime are affected by the wearout-awareness. For comparing the wearout, we measure the CV (Coefficient of Variation)[1] across the amount of data written to each PIM device, after an application completes its execution.

Figure 7a shows that, for all tested cases, CV values noticeably decrease when the wearout-awareness is enabled. This confirms that DY-LR with the wearout-awareness can effectively balance wearout levels across the array by avoiding the use of the PIM device with the highest wearout level. This will eventually prevent an early retirement of a particular PIM device from an unbalanced use. We also observe that enabling wearout-awareness in Case 3 outweights the benefits in Case 2, due to the heavy skewness of the initial data distribution in Case 3 (Table 2). Figure 7b shows that the runtime overhead of the wearout-awareness is 10% on average. This is because, with the

[1] The CV is defined as the ratio of the standard deviation s to the mean m of written data sizes across PIM devices. CV = $\frac{s}{m}$.

wearout-awareness, the AnalyzeThat runtime has to periodically collect internal PIM-specific information (e.g., current wearout status) from each PIM. Also, note that the wearout-aware DY-LR performs comparable to other data placement policies that are not wearout-aware (i.e, RR, HS and LA). For instance, in Case 3 with WordCount (WC), the wearout-aware DY-LR outperforms LA-LR, HS-LR, and RR-LR by 1.8%, 8.1%, and 30%, respectively, with seven PIM devices (PIM(7)).

8 Conclusion

PIM architectures can offer several advantages to data analysis applications. However, there exists a high entry barrier to programmers because of the complexity of the hardware and the various new aspects it offers (e.g., data placement, wearout-level, etc.), which need to be considered to take advantage of the architecture. We have developed AnalyzeThat as a means to abstract such manual efforts, and to provide an easy and efficient programming platform to users. Specifically, it provides a high-level data and programming abstraction, PADS (PIM-aware data structure), which allows users to expose the PIM devices as a key-value container, and apply a set of operations on the container. The PADS abstraction further provides a runtime system that collects internal information from the PIM devices, and makes intelligent decisions such as dynamic data placement on the PIM array. We have shown how representative data analysis applications can be effectively implemented using PADS. Our evaluation of AnalyzeThat shows that it is viable, and can be used to develop complex data analysis applications atop the PIM array.

Acknowledgements This research was supported in part by the U.S. DOE's Office of Advanced Scientific Computing Research (ASCR) under the Scientific data management program, and the National Research Foundation of Korea (NRF) Grant funded by the Korea Government (MSIP) (No. 2015R1C1A1A0152105). The work was also supported by, and used the resources of, the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at ORNL, which is managed by UT Battelle, LLC for the U.S. DOE, under the contract No. DE-AC05-00OR22725.

References

1. Kogge, P.M., Brockman, J.B., Sterling, T., Gao, G.: Processing in memory: chips to petaflops. In: Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA, vol. 97 (1997)
2. Murphy, R.C., Kogge, P.M., Rodrigues, A.: The characterization of data intensive memory workloads on distributed PIM systems. In: Intelligent Memory Systems (2001)
3. Adibi, J., Barrett, T., Bhatt, S., Chalupsky, H., Chame, J., Hall, M.: Processing-in-memory technology for knowledge discovery algorithms. In: Proceedings of the DaMoN (2006)
4. Brockman, J.B., Thoziyoor, S., Kuntz, S.K., Kogge, P.M.: A low cost, multithreaded processing-in-memory system. In: Proceedings of the WMPI (2004)
5. Kang, Y., Huang, W., Yoo, S.-M., Keen, D., Ge, Z., Lam, V., Pattnaik, P., Torrellas, J.: FlexRAM: toward an advanced intelligent memory system. In: Proceedings of the ICCD (2012)
6. Draper, J., Chame, J., Hall, M., Steele, C., Barrett, T., LaCoss, J., Granacki, J., Shin, J., Chen, C., Kang, C.W. et al.: The architecture of the DIVA processing-in-memory chip. In: Proceedings of the SC (2002)
7. Pugsley, S.H., Jestes, J., Zhang, H., Balasubramonian, R., Srinivasan, V., Buyuktosunoglu, A., Davis, A., Li, F.: NDC: analyzing the impact of 3D-stacked memory logic devices on MapReduce workloads. In: Proceedings of the ISPASS (2014)
8. Scrbak, M., Islam, M., Kavi, K.M., Ignatowski, M., Jayasena, N.: Processing-in-Memory: Exploring the Design Space. Springer, Cham (2015)
9. Zhang, D., Jayasena, N., Lyashevsky, A., Greathouse, J.L., Xu, L., Ignatowski, M.: TOP-PIM: throughput-oriented programmable processing in memory. In: Proceedings of the HPDC (2014)
10. Micron's Automata: https://www.micronautomata.com
11. Raoux, S., Burr, G.W., Breitwisch, M.J., Rettner, C.T., Chen, Y.-C., Shelby, R.M., Salinga, M., Krebs, D., Chen, S.-H., Lung, H.-L.: Phase-change random access memory: a scalable technology. IBM J. Res. Dev. **52**(4), 5 (2008)
12. Strukov, D.B., Snider, G.S., Stewart, D.R., Williams, R.S.: The missing memristor found. Nature **453**, 7191 (2008)
13. Driskill-Smith, A.: Latest advances and future prospects of STT-RAM. In: Proceedings of the NVMW (2010)
14. Islam, M., Scrbak, M., Kavi, K.M., Ignatowski, M., Jayasena, N.: Improving node-level MapReduce performance using processing-in-memory technologies. In: Proceedings of the Euro-Par (2014)
15. Zhang, D.P., Jayasena, N., Lyashevsky, A., Greathouse, J., Meswani, M., Nutter, M., Ignatowski, M.: A new perspective on processing-in-memory architecture design. In: Proceedings of the SIGPLAN, ser. MSPC '13, pp. 7:1–7:3 (2013)
16. Dongarra, J.: The international exascale software project roadmap. Int. J. High Perform. Comput. Appl. **25**, 3–60 (2011)
17. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.: SkewTune: mitigating skew in MapReduce applications. In: Proceedings of the SIGMOD (2012)
18. Yoo, R.M., Romano, A., Kozyrakis, C.: Phoenix rebirth: scalable MapReduce on a large-scale shared-memory system. In: Proceedings of the IISWC (2009)
19. Lee, S., Sim, H., Kim, Y., Vazhkudai, S.S.: Analyzethat: a programmable shared-memory system for an array of processing-in-memory devices. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE Press pp. 619–624 (2017)
20. Scrbak, M., Islam, M., Kavi, K.M., Ignatowski, M., Jayasena, N.: Processing-in-memory: exploring the design space. In: Proceedings of the ARCS (2015)
21. OpenCL: The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/
22. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
23. Hadoop, A.: Apache hadoop. http://hadoop.apache.org (2011)

24. Talbot, J., Yoo, R.M., Kozyrakis, C.: Phoenix++: modular MapReduce for shared-memory systems. In: Proceedings of the MapReduce (2011)
25. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a MapReduce framework on graphics processors. In: Proceedings of the PACT (2008)
26. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the USENIX, vol. 10 (2010)
27. Pugsley, S.H., Jestes, J., Balasubramonian, R., Srinivasan, V., Buyuktosunoglu, A., Li, F., et al.: Comparing implementations of near-data computing with in-memory MapReduce workloads. IEEE Micro **34**(4), 1 (2014)
28. Cache Coherent Interconnect for Accelerators (CCIX): http://www.ccixconsortium.com
29. Nobis, S.: AMD's Unified CPU & GPU Processor Concept
30. Loh, G., Jayasena, N., Oskin M. et al.: A processing in memory taxonomy and a case for studying fixed-function PIM. In: Near-Data Processing Workshop (2013)
31. ARM Cortex-A5: http://www.arm.com/products/processors/cortex-a/cortex-a5.php
32. Netezza Data Warehouse | IBM: https://www.ndm.net/datawarehouse/IBM/netezza
33. Fang, J., Varbanescu, A.L., Sips, H.: A comprehensive performance comparison of CUDA and OpenCL. In: 2011 International Conference on Parallel Processing (2011)
34. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of the ACM SIGOPS, vol. 41, no. 6 (2007)
35. Debnath, B., Sengupta, S., Li, J.: FlashStore: high throughput persistent key-value store. In: Proceedings of the VLDB Endowment, vol. 3, no. 1–2 (2010)
36. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. Stanford InfoLab, Stanford (1999)
37. EnWiki.NET: Encyclopaedia Britannica Ultimate. http://www.enwiki.net/
38. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data (2014)
39. Adamic, L.A., Huberman, B.A.: Power-law distribution of the world wide web. Science **287**(5461), 2115 (2000)

Sangkeun Lee received the B.S. degree in computer science from Korea Advanced Institute of Science and Technology in 2005 and the Ph.D. degree in computer science and engineering from Seoul National University in 2012. He joined Oak Ridge National Laboratory in 2013. His research interests include large-scale graph mining and analytics, big data systems and architectures, information retrieval and recommender systems and their applications.

Hyogi Sim received a M.S. in computer sciencefrom Virginia Tech in 2014 and is currently pursuinghis Ph.D. degree at Virginia Tech. He alsoearned a M.S. in Computer Engineering and aB.S. in Civil Engineering from Hanyang Universityin South Korea. He joined Oak Ridge NationalLaboratory in 2015, as a post-masters associate.During this appointment, he conductedresearch and development on active storagesystems and scientific data management forHPC systems. He is currently an HPC systemsengineer in Oak Ridge National Laboratory. His primary role is to designand develop a checkpoint-restart storage system for the exascalecomputing project. His areas of interest include storage systems anddistributed systems.

Youngjae Kim received his Ph.D. degree in Computer Science and Engineering from Pennsylvania State University, University Park, PA, USA in 2009. He is currently an assistant professor in the department of computer science and engineering at Sogang University, Seoul, Republic of Korea. Before joining Sogang University, Dr. Kim was a staff scientist in the U.S. Department of Energy's Oak Ridge National Laboratory (2009–2015) and an assistant professor in Ajou University, Suwon, Republic of Korea (2015–2016). Dr. Kim received the B.S. degree in computer science from Sogang University, Republic of Korea in 2001, and the M.S. degree from KAIST in 2003. His research interests include distributed file and storage, parallel I/O, operating systems, emerging storage technologies, and performance evaluation.

Sudharshan S. Vazhkudai leads the TechnologyIntegration (TechInt) group in the NationalCenter for Computational Sciences (NCCS) atOak Ridge National Laboratory (ORNL). NCCShosts the Oak Ridge Leadership Computing Facility(OLCF), which is home to the 27 petaflopsTitan supercomputer. Dr. Vazhkudai leads agroup of 17 HPC researchers and systems softwareengineers; the group is charged with deliveringnew technologies into OLCF by identifyinggaps in the system software/hardware stack, anddeveloping, hardening and deploying solutions. His group's technologyscope includes the deep-storage hierarchy, non-volatile memory, systemarchitecture, system monitoring, and data and metadata management.Dr. Vazhkudai received a Ph.D. and Masters in Computer Science fromthe University of Mississippi in 2003 and 1998 respectively.

Affiliations

Sangkuen Lee[1] · Hyogi Sim[1] · Youngjae Kim[2] · Sudharshan S. Vazhkudai[1]

[1] Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, USA

[2] Department of Computer Science and Engineering, Sogang University, Office: AS911, 35 Baekbeomro, Mapogu, Seoul 04107, Republic of Korea