Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

# Improving Hadoop MapReduce performance on heterogeneous single board computer clusters[☆]

Sooyoung Lim [a], Dongchul Park [b,*]

[a] Division of Computer Science, Sookmyung Women's University, Seoul 04310, South Korea
[b] Department of Industrial Security, Chung-Ang University, Seoul 06974, South Korea

ARTICLE INFO

*Keywords:*
Big data
Hadoop
MapReduce
YARN
Heterogeneous cluster
Raspberry Pi
Single board computer (SBC)

ABSTRACT

Over the past decade, Apache Hadoop has become a leading framework for big data processing. Single board computer (SBC) clusters, predominantly adopting Raspberry Pi (RPi), have been employed to explore the potential of MapReduce processing in terms of low power and cost because, capital costs aside, power consumption has also become a primary concern in many industries. After building SBC clusters, it is prevalent to consider adding more nodes, particularly newer generation SBCs, to the existing clusters or replacing old (or inactive) nodes with new ones to improve performance, inevitably causing heterogeneous SBC clusters. The Hadoop framework on these heterogeneous SBC clusters creates challenging new problems due to computing resource discrepancies in each node. Native Hadoop does not carefully consider the heterogeneity of the cluster nodes. Consequently, heterogeneous SBC Hadoop clusters result in significant performance variation or, more critically, persistent node failures. This paper proposes a new Hadoop Yet Another Resource Negotiator (YARN) architecture design to improve MapReduce performance on heterogeneous SBC Hadoop clusters with tight computing resources. We newly implement two main scheduling policies on Hadoop YARN based on the correct computing resource information that each SBC node provides: (1) two (master-driven *vs.* slave-driven) MapReduce task scheduling frameworks to determine more effective processing modes and (2) ApplicationMaster (AM) and reduce task distribution mechanisms to provide the best Hadoop performance by minimizing performance variation. Thus, the proposed Hadoop framework makes the best use of the performance-frugal SBC Hadoop cluster by intelligently distributing MapReduce tasks to each node. To our knowledge, the proposed framework is the first redesigned Hadoop YARN architecture to address various challenging problems particularly on *heterogeneous SBC Hadoop clusters* for big data processing. The extensive experiments with Hadoop benchmarks demonstrate that the redesigned framework performs better performance than the native Hadoop by an average of 2.55× and 1.55× under I/O intensive and CPU-intensive workloads, respectively.

## 1. Introduction

The Apache Hadoop platform has been widely employed in many industries and academic fields for over a decade as the primary big data storage and processing framework [1–3]. Many companies have adopted powerful, general-purpose servers and clusters as Hadoop workhorses to use the big data processing software frameworks best. Moreover, to further improve Hadoop application performance, they leveraged Graphics Processing Units (GPUs) for extra computation and InfiniBand for high-speed network connection [4–7]. However, due to their high cost, not all industries or academic fields want (or can afford) such powerful computing nodes or clusters that can do everything reasonably well. Recently, energy efficiency has become another crucial factor in many industries. Thus, microservers have been a focus for energy saving and purpose-built computing [8]. A microserver is a small server appliance that integrates all server motherboard functions onto a single microchip (i.e., the system on a chip) and crams many energy-frugal microchips into a single rack. Thus, the power consumption of a microserver is far below that of a high-end general-purpose server system [8]. An accelerator or a new computing model is another solution to address energy efficiency-related challenges. Some studies have adopted Field Programmable Gate Arrays (FPGAs) [9] and a new energy-efficient computing model, such as In-Storage Computing

☆ This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP), South Korea under the Convergence security core talent training business support program (IITP-2024-RS-2023-00266605) grant funded by the Korea government (MSIT).

* Corresponding author.
*E-mail addresses:* slim@sookmyung.ac.kr (S. Lim), dongchul@cau.ac.kr (D. Park).

https://doi.org/10.1016/j.future.2024.06.025
Received 7 November 2023; Received in revised form 30 April 2024; Accepted 13 June 2024
Available online 15 June 2024

(ISC) [10], to reduce power consumption in Hadoop MapReduce applications.

A single board computer (SBC) is an emerging platform to build low-power, low-cost computing systems [11]. An SBC is a complete, functioning minicomputer with a microprocessor, input/output (I/O) ports, main memory, and other features built on a single circuit board the size of a credit card [12]. This simplistic design can be antithetical to the current multipurpose personal computers (PCs). However, such simplicity makes SBCs perfect as embedded computers adopted to operate more complex devices, such as those used in consumer, automotive, and even military applications [13]. The SBCs' energy efficiency and low cost enable them to fill niche applications by bridging the performance gap between controller boards and PCs [14].

Among SBCs, Raspberry Pi (RPi) has dominated the market since it was first released in 2012. Now, RPi has become the third best-selling general-purpose computer worldwide [8], following the Apple Mac and Windows PC [15,16]. They did not have sufficient computing resources, so early generations of RPi models were primarily used for hands-on education, such as learning programming languages or orchestrating network management. However, as RPi evolved into a more powerful minicomputer with multicore CPUs, more memory capacity, and higher-speed I/O ports, many researchers have built RPi clusters and examined the possibility of operating heavy applications including massive data processing or edge computing [11,17]. Lee et al. [8] recently demonstrated that the latest generation RPi 4B-based clusters can provide sufficient computational performance to operate heavy big data applications effectively (i.e., more than hundreds of gigabytes).

In building SBC clusters, it is prevalent to consider expanding or reconfiguring the clusters in the future by adding more SBC nodes, particularly newer generation SBCs, into the existing clusters or replacing old (or incorrectly working) nodes with new ones to improve performance. This approach causes heterogeneous SBC clusters. Hadoop MapReduce processing on heterogeneous SBC clusters creates challenges because each node has discrepant computing resources. Moreover, a native Hadoop platform does not carefully consider the heterogeneity of SBC cluster nodes. Heterogeneous SBC Hadoop clusters exhibit significantly unstable performance depending on the MapReduce task scheduling policies. Importantly, it also frequently fails to complete the job because weaker nodes (i.e., nodes with lower computational capabilities) are overburdened with excessive task processing.

This paper proposes a new Hadoop Yet Another Resource Negotiator (YARN) architecture design to improve Hadoop MapReduce application performance on heterogeneous SBC Hadoop clusters. A native Hadoop framework does not carefully consider cluster node heterogeneity; thus, many research studies have tried to address this problem on typical Hadoop clusters with sufficient computational capabilities. However, to our knowledge, this research is the first to investigate challenging problems on heterogeneous, power-frugal SBC Hadoop clusters for big data processing. Extensive studies and experiments found the following critical challenges of the native Hadoop framework on heterogeneous RPi clusters.

**Low computational capability**: Previous generation RPis (i.e., earlier than the fourth generation) have only up to 1 GB of memory. These RPi nodes cannot successfully process numerous data in the native Hadoop framework. They turn into inactive nodes and eventually stop working. Adding more powerful RPi nodes, such as RPi 4B, to the existing cluster cannot fundamentally resolve this problem. The native Hadoop framework does not provide a mechanism to carefully detect the computational capabilities of each node in the cluster; thus, it keeps assigning excessive tasks even to such weak SBC nodes. The native Hadoop only provides user configurations, which relies on users' empirical insight. More importantly, this user-level Hadoop configuration cannot resolve this problem. To overcome this inherent limitation, we redesigned a Hadoop NodeManager (NM) component to correctly identify the computing resources of each RPi node in the cluster. Thus, the redesigned Hadoop carefully investigates each node's computational capabilities in advance and *automatically* assigns appropriate (i.e., different) numbers of tasks to them without user configurations.

**Considerable performance variation**: Unlike powerful computer-based heterogeneous Hadoop clusters, resource-frugal SBC-based heterogeneous Hadoop clusters exhibit a considerable performance variation for each job execution. This inconsistent Hadoop performance primarily originates from the performance gap between heterogeneous SBC cluster nodes. Our comprehensive study found that a reduce task assignment policy also significantly influences the overall Hadoop performance, particularly in SBC Hadoop clusters. A native Hadoop framework does not provide specific optimal policies to distribute the Hadoop ApplicationMaster (AM) and reduce tasks to the appropriate cluster nodes effectively. Depending on their locations, Hadoop MapReduce applications exhibit tremendously different overall performance, resulting in inconsistent performance for each job execution. To provide the best Hadoop MapReduce performance on heterogeneous RPi clusters consistently, we propose optimal AM and reduce task distribution policies. Consequently, the newly designed Hadoop framework offers the best consistent performance.

The main contributions of this paper are as follows:

- **Redesigned Hadoop YARN Architecture**: We implement two primary Hadoop YARN scheduling policies. First, two MapReduce task scheduling mechanisms are proposed to determine more effective Hadoop processing modes (i.e., parallel processing vs. sequential processing) based on the computational capabilities of each node. Second, new AM and reduce task distribution policies are implemented to offer the best overall performance by carefully assigning them to the appropriate cluster nodes.
- **Extensive Performance Evaluation**: We propose two Hadoop frameworks (i.e., master-driven and slave-driven) based on task scheduling policies. We compare them to a native Hadoop design, adopting an extra swap space allocation mechanism. The extensive experiments with Hadoop benchmarks demonstrate that the redesigned framework achieves an average of 2.55× and 1.55× performance improvement under the I/O-intensive (e.g., Terasort) and CPU-intensive (Wordcount) workloads, respectively.
- **Challenges and Suggestions**: Big data processing on the SBC clusters with limited resources introduces unexpected challenges (e.g., temporary data and uberized jobs). We investigate these challenges and provide suggestions. We also discuss the Hadoop MapReduce performance on SBC clusters in terms of performance-per-watt and per-dollar to explore possibilities in the future.

The rest of this paper is organized as follows. Section 2 presents related work. Next, Section 3 details the design challenges and the proposed solutions. Then, Section 4 provides the comprehensive experimental results and analyses. Section 5 discusses the diverse challenging problems and suggestions, and Section 6 presents the conclusion.

## 2. Background and related work

This section explains RPi and Apache Hadoop in more detail. In addition, it discusses the related work.

### 2.1. Raspberry Pi

Raspberry Pi is a cost-effective, energy-efficient computing platform. Initially, it was intended to promote teaching computer science in schools by inspiring students to engage in electronics and programming [11]. Now, it is also a preferred embedded controller in numerous industries, such as those using consumer, smart home, automotive, medical, and even military applications [13].

Table 1 compares the hardware specifications of the third generation RPis (RPi 3B and 3B+) and the fourth generation RPi (RPi

**Table 1**
Specifications of the third and fourth generation Raspberry Pi [8]. Here, 'Power' stands for power consumption. Our heterogeneous SBC cluster comprises these three types of RPi nodes.

| | Raspberry Pi 3B | Raspberry Pi 3B+ | Raspberry Pi 4B |
|---|---|---|---|
| CPU | ARM Cortex-A53 @ 1.2 GHz (4 cores) | ARM Cortex-A53 @ 1.4 GHz (4 cores) | ARM Cortex-A72 @ 1.5 GHz (4 cores) |
| RAM | 1 GB LPDDR2 SDRAM | 1 GB LPDDR2 SDRAM | 1,2,4, or 8 GB LPDDR4 SDRAM |
| Ethernet | 100 Mbps Ethernet | 300 Mbps Gigabit Ethernet | Native Gigabit Ethernet |
| USB | 4× USB 2.0 | 4× USB 2.0 | 2× USB 3.0 + 2× USB 2.0 |
| Power | 1.4W(idle), 3.7W(full-load) | 1.9W(idle), 5.1W(full-load) | 2.7W(idle), 6.4W(full-load) |
| Release | February 2016 | March 2018 | June 2019 |
| Price | $35 | $35 | $35(1 GB), $45(2 GB), $55(4 GB) |

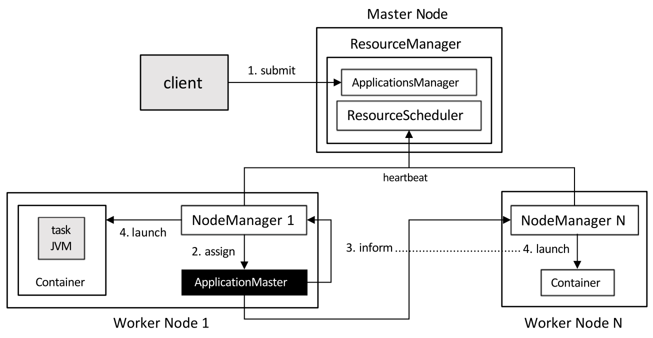


**Fig. 1.** Overall Hadoop YARN architecture and service flow. *Gray* represents components outside the YARN architecture.

4B). Particularly, the computing power of RPi 4B is unprecedentedly improved due to its historical full-chip redesign, including 2.5× faster four core CPUs, the first graphics processor upgrade, the first UBS 3.0 port, full speed 1 GB Ethernet, an HDMI port for 4 K display, and up to 8 GB of RAM [18]. Thus, Hadoop performance on the RPi 4B cluster exhibited a performance about 2× higher than the RPi 3B cluster [8].

## 2.2. Apache Hadoop YARN

Apache Hadoop YARN, also known as MapReduce v2, is a resource management and job scheduling technology in the open source Hadoop framework [19]. It was introduced to address the following limitations of the original MapReduce framework: inefficient failure handling and the fault tolerance mechanism. To resolve these limitations, YARN separates resource management and job scheduling from the application execution layer. It manages each MapReduce task using a logical bundle unit called a container [20].

### 2.2.1. Architecture

Apache Hadoop YARN consists of three key components: (1) *ResourceManager* (RM), (2) *NodeManager* (NM), and (3) *ApplicationMaster* (AM).

The RM is a Java process running on a master node. It manages and allocates computing resources such as CPU and memory across Hadoop cluster for diverse jobs. The RM has two main components: *ResourceScheduler* and *ApplicationsManager*. The *ResourceScheduler* in YARN is responsible for job scheduling based on policies and priorities. The *ApplicationsManager* is an interface maintaining application lists. It accepts job submission to YARN, negotiates the first container for executing AM, and manages restarting the AM container on failure. After application submission, the *ApplicationsManager* first validates the application's specifications. If the application requests unsatisfiable resources, it denies the application, otherwise, *ApplicationMaster* launcher launches the AM.

The NM is a process that runs on a worker node in a Hadoop cluster. Each worker node has an NM managing MapReduce task execution on that node. The NM periodically sends a heartbeat to the RM to provide node status information, and launches and manages (i.e., executes and monitors) containers on the worker node based on the instructions from the RM.

The AM executes the main function of an application and orchestrates an application's operation within the cluster. It negotiates with the RM to obtain resources for launching containers to execute tasks. The AM manages all task execution processes including assigning tasks to containers, tracking each task execution status, monitoring the progress, and handling task failures.

### 2.2.2. YARN service flow

Fig. 1 presents the overall YARN service flow. A MapReduce process begins when a client submits a job to the Hadoop cluster. The client communicates with the RM to submit the job. After receiving the job submission, the RM designates a specific node to host the AM and notifies the corresponding node's NM. Then, the AM is launched within a container on that node. After the AM starts, it interacts with NMs to request containers. The NM allocates resources (CPU and memory) to containers and launches them according to the AM's request. These containers host the actual MapReduce tasks. Each MapReduce task is executed as an independent process outside the NM to achieve fault tolerance and continuous task execution without affecting the NM loss. We redesigned the whole YARN scheduling mechanism to improve Hadoop MapReduce performance on heterogeneous SBC clusters.

## 2.3. Related work

### 2.3.1. SBC Hadoop clusters

Kaewkasi and Srisuruk [21] built a Hadoop cluster of 22 cubieboards. Each node is an ARM Cortex A8 processor-based SBC. They installed Apache Spark on the cluster and conducted Spark experiments over the Hadoop distributed file system (HDFS). The authors claimed the cluster could successfully process a 34 GB Wikipedia article file in an acceptable time and consumed the power of 0.061 to 0.322 kWh for all benchmarks. They concluded that the cluster performance bottleneck primarily originated from low I/O and CPU processing capabilities.

Qureshi and Koubaa [12] built two SBC clusters (RPi 2B and Odroid XU-4 platforms) of 20 nodes to address the energy efficiency problems of a small data center. They deployed Apache Hadoop on the clusters and evaluated their performance regarding task execution time, memory utilization, network throughput, and power consumption. They claimed the RPi cluster exhibited lower performance than other clusters (Odriod UX-4 cluster of 20 nodes and a regular PC cluster of four nodes). However, this is predominantly because they adopted a very early generation RPi.

Recently, Lee et al. [8] conducted comprehensive Hadoop performance research on the latest generation RPi cluster (i.e., the RPi 4B cluster). They set up a Hadoop cluster of five RPi 4B nodes (4 GB of RAM each) and extensively performed Apache Hadoop and Spark benchmarks. They also investigated various storage media performance effects on the overall Hadoop cluster performance by leveraging the USB 3.0 interface. They demonstrated that the latest generation RPi 4B cluster offers sufficient performance to process massive data (e.g., 500 GB to 1TB, not just a few hundred megabytes) successfully [22]. The authors concluded that RPi 4B finally achieved unprecedented performance improvement and that the RPi 4B cluster exhibited a considerable potential to process *actual* big data effectively.

### 2.3.2. Improving hadoop cluster performance

Speculative execution is Hadoop's fault tolerant mechanism to address straggler nodes in the cluster. This slowdown occurs for various reasons, such as hardware problems, network congestion, or other resource bottlenecks. As a result, some tasks may take longer, causing a significant job completion delay. Some studies have developed efficient task scheduling algorithms considering the computing performance gap to resolve this problem [23–26].

The Hadoop framework uses data locality to reduce network traffic. The Hadoop master node attempts to schedule a task on a node that contains the corresponding input data. Data locality ensures that each task is assigned to the node where the required data are stored, which can be time-consuming and resource-intensive. For example, tasks assigned to straggler nodes may noticeably delay the job completion time. Some researchers have proposed data locality improvement mechanisms to solve this problem [27–29].

Computing resources, such as CPU and memory, are another crucial factor for designing Hadoop schedulers. Hadoop nodes in the data center clusters receive a large number of jobs and require more resources to execute them, causing the race condition. Some research studies have improved the cluster resource utilization [30–33]. Heterogeneous clusters raise different problems of improving Hadoop cluster performance. Most researchers have classified cluster nodes according to their computing capacities and have proposed diverse deadline-aware mechanisms [34–36].

Recently, integrating an accelerator into the distributed platform has been studied to accelerate task computation. For instance, the FPGA-accelerated heterogeneous architecture based on YARN provides resource management and programming support for computing-intensive applications by adopting FPGAs [37].

However, most existing studies have been conducted on *general (i.e., powerful)* PC (or server)-based Hadoop clusters, not *resource-frugal heterogeneous* SBC clusters. Heterogeneity on low-power, low-cost SBC Hadoop clusters raises vastly different challenging problems. Thus, most existing studies are not applicable to our SBC cluster environment. This paper explores novel challenges in the heterogeneous SBC Hadoop cluster.

## 3. Main design

This section describes the redesigned Hadoop YARN architecture for a heterogeneous RPi Hadoop cluster.

### 3.1. Motivations

The MapReduce job submission requires the DataNode Java Virtual Machine (JVM) for input data split management and NodeManager (NM) JVM for cluster resource negotiation. After the MapReduce job starts, the ResourceManager (RM) assigns the ApplicationMaster (AM) to one of the worker nodes in the cluster. Each (map or reduce) task runs on its task JVM. Each JVM execution requires a certain level of memory capacity. However, early generation RPis (before the fourth generation) do not meet the MapReduce task requirements, causing an *out-of-memory* (OOM) problem.

Fig. 2 presents the OOM problem for Hadoop MapReduce task execution on the third generation RPi (i.e., RPi 3B+) worker node. A native Hadoop runs on an RPi 3B+ node with two map tasks and 128 MB data configurations. We examined the system resource information when the OOM occurs. As in Fig. 2-(a), the total memory usage suddenly drops. Although the map task is fully completed because each task is executed independently, NM is interrupted when the map task count exceeds two. This NM loss results in a job execution halt because NM is responsible to the reduce tasks for negotiating resource allocation. Fig. 2-(b) illustrates the CPU utilization of the RPi 3B+ node. The CPU utilization soars to about 90% when the OOM problem appears because of context switching overhead in each JVM. We observed that adjusting the virtual memory allocation ratio does not work to solve this problem.

For verification, we built another native Hadoop node on the latest fourth generation RPi (RPi 4B with 1 GB of RAM). We performed identical experiments with Hadoop MapReduce configurations identical to RPi 3B+. As described in Section 2.1, although RPi 4B has much more robust computational capabilities than RPi 3B+, the OOM problem still

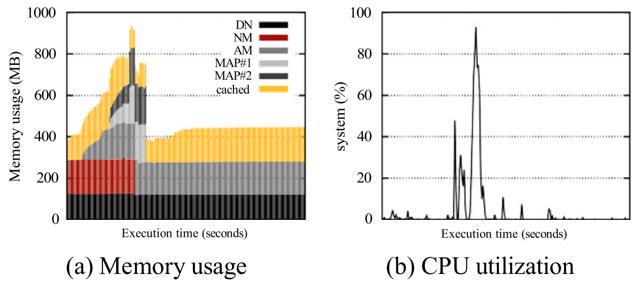


**Fig. 2.** Resource usage over time on an RPi 3B+ worker node for processing 128 MB of data with two map tasks.

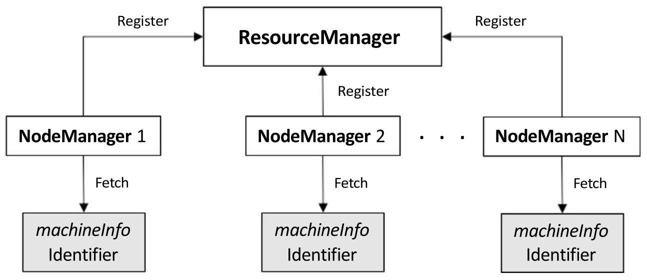


**Fig. 3.** The *machineInfo* identifier component. In each worker node, NodeManager fetches the correct computing resource information from the *machineInfo* component and sends it to the ResourceManager for registration.

exists on the RPi 4B node, implying that the physical memory capacity critically influences the MapReduce job execution. We observed that a native Hadoop does not allow two or more concurrent map task execution instances on a node with 1 GB of RAM, which is the main reason the RPi 3B+ (and even 4B) Hadoop node with 1 GB of RAM cannot complete substantial data processing (requiring two or more concurrent map tasks). Please note that RPi 4B here has the same RAM amount (1 GB) as RPi 3B+ for verification.

We can simply assign an additional swap space to the storage for a logical memory space extension to resolve this problem naively [38, 39]. However, applying this approach to low-performance RPi nodes, such as RPi 3B (or 3B+), creates another problem: low I/O performance causes an overall Hadoop performance bottleneck. Our experiments comprehensively investigate this problem (see Section 4.2.4).

Instead of an additional swap space assignment, we resolve this critical problem by redesigning the core scheduling mechanisms for Hadoop YARN. Since multiple concurrent map task execution on a memory-frugal RPi node causes Hadoop MapReduce job failure, the redesigned Hadoop YARN automatically changes the MapReduce task scheduling policy from parallel processing to sequential processing mode. That is, when the proposed Hadoop framework detects a memory-frugal node in the cluster, it dynamically switches the task scheduling policy to the single map task mode to prevent concurrent map task execution on such weak nodes. Consequently, the redesigned Hadoop YARN framework enables even a resource-scanty SBC node to complete the MapReduce job successfully.

### 3.2. Design space and implementation

This section presents several design challenges and respective implementations.

#### 3.2.1. Computational capability identification

A native Hadoop framework does not provide a mechanism to identify the actual computational capabilities of each node, such as a physical memory space. As described, the physical memory capacities of nodes critically affect the concurrent MapReduce task execution.

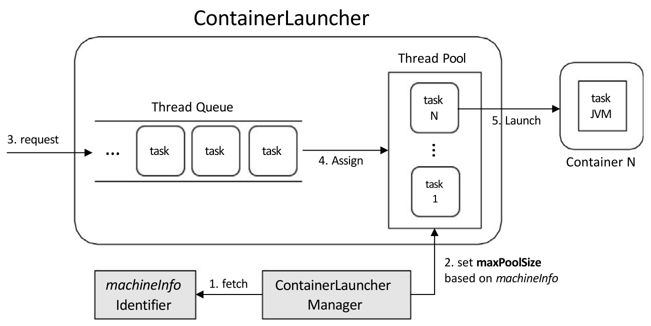


**Fig. 4.** Redesigned *ContainersLauncher* component in NodeManager for our slave-driven design.

To implement the redesigned YARN scheduler, we must first design another component on each worker node, *machineInfo*, that identifies each node's computing resources and sends the information to YARN. Fig. 3 presents the proposed *machineInfo* component, a Linux child process of NM. In each worker node, NM fetches actual computing resource information from the *machineInfo* identifier component and sends it to RM. The proposed *machineInfo* component is executed only once during the NM service setup.

Since the physical resource specifications of each node are not generally changed while the system is running, our proposed design does not need to periodically transmit the *machineInfo* information to the ResourceManager (RM). Thus, we did not adopt the piggybacking mechanism using periodic heartbeat messages, which reduces additional communication overhead. Instead, using a protocol buffer API, we implemented a novel transmission standard, including *machineInfo*. This modified inter-process communication (IPC) protocol enables each NM to transmit *machineInfo* to RM. While each NM's information is registered on RM, this *machineInfo* must be included. The RM manages each NM's information using ResourceManagerNode (RMNode) component; hence, each RMNode is created in RM whenever each NM is registered on RM. Thus, we added each NM's *machineInfo* information into the corresponding RMNode properties. Our precise experiment (using *jmap* command with *-histo* option) demonstrated that this design does not cause a Hadoop cluster scalability problem because each RMNode's overhead is negligible in terms of memory consumption (i.e., just additional 8 bytes for each single RMNode instance).

Based on the information that each *machineInfo* identifier component provides, the redesigned YARN framework dynamically switches to the sequential processing or conventional parallel processing mode accordingly.

### 3.2.2. *Task scheduling: Master-driven vs. Slave-driven*

We propose two task scheduling frameworks to manage MapReduce tasks effectively depending on the node type: master-driven and slave-driven frameworks. Both provide an automatic Hadoop configuration mechanism based on *machineInfo* when the YARN service starts. The master-driven framework is associated with the RM. In contrast, the slave-driven framework adjusts the number of threads in the NM.

- **Master-driven Framework**: After accepting each NM's registration request, the RM promptly calculates the number of available containers for each worker node based on container-related properties in Hadoop YARN configurations (see Table 3). Unlike the native Hadoop, which requires users' experience and insight to configure properties effectively, the proposed framework can automatically configure the appropriate number of concurrent containers for each worker node based on *machineInfo*. If SBC worker nodes have powerful computational capabilities, such as RPi 4B, the framework adopts the Hadoop default settings (i.e., the parallel processing mode); otherwise, it allocates a single MapReduce task at a time via RM's *ResourceScheduler* (i.e., the sequential processing mode).
- **Slave-driven Framework**: Each NM has a component that launches the assigned containers in NM, called *ContainersLauncher* (see Fig. 4). This component is initialized when the NM service starts. It has a thread pool, called *ContainerLauncher*, to launch each container. Each container is executed within a single thread. In native Hadoop, this thread pool uses the *newCachedThreadPool* component to create new threads as needed. Thus, the native Hadoop creates more than two concurrent tasks even on the memory-frugal SBC nodes, such as 3rd generation RPis, causing a Hadoop job failure. Our redesigned *ContainersLauncher* component intelligently manages the maximum number of concurrent threads for each SBC node based on *machineInfo*. If an SBC worker node is identified as a memory-frugal node, this slave-driven framework sets the maximum thread count to one by implementing the *newFixedThreadPool* component (i.e., the sequential processing mode). This newly designed thread pool adopts *LinkedBlockingQueue* for the wait queue and arranges elements in an FIFO order. This means each *ContainerLaunch* thread for the container execution request waits in this wait queue of the thread pool sequentially. Thus, each thread in the wait queue does not contend for computing resources and a starvation problem does not happen. On the other hand, if the SBC node is identified as a powerful node, this framework employs the Hadoop default settings (i.e., the parallel processing mode).

Unlike the master-driven design, the slave-level framework executes those task scheduling configurations inside NM (i.e., the slave node). In addition, it does not override each worker node's preset values of container-related properties. Thus, the RM attempts to allocate multiple concurrent containers even to memory-frugal SBC nodes. However, the slave-driven design limits the available container count to one in the weak SBC nodes, and the remaining containers remain pending in the thread pool's waiting queue. Therefore, the master-driven design focuses more on a fast overall execution time because the RM globally manages the MapReduce task assignment and (if possible) attempts to assign more tasks to powerful nodes. In contrast, the slave-driven scheduler locally determines each task execution on each worker node, although the RM globally assigns tasks to worker nodes. Hence, the proposed slave-driven design somewhat considers task load balancing and total execution time. Our extensive experiments evaluate these two designs in more detail (Section 4.2.5).

### 3.2.3. *ApplicationMaster and reduce task distribution*

Performance-frugal SBC-based Hadoop clusters exhibit a considerable performance variation for each job execution. This inconsistent Hadoop application performance fundamentally results from performance gap between heterogeneous SBC nodes. The native Hadoop framework does not carefully consider this heterogeneity problem because it does not provide explicit data locality policies for both AM and reduce tasks. This subsection presents our new data locality policies.

Unlike map tasks, no specific rules exist for assigning the AM and the reduce tasks in cluster nodes. That is, the AM and reduce tasks can be assigned to any node in the cluster. Thus, according to their distributed locations, significant performance variations occur because of node performance discrepancy. For instance, if the reduce tasks are assigned to any SBC node with low computational capability, the Hadoop MapReduce job takes noticeably longer to complete because map tasks on these nodes cannot take full advantage of the data locality.

Fig. 5-(a) presents native Hadoop task execution priorities for each task request. Hadoop tasks have three levels of data locality, and Hadoop prioritizes tasks for data processing optimization in the following order: NODE_LOCAL (highest priority), RACK_LOCAL (intermediate priority), and OFF_SWITCH (lowest priority). These locality preference

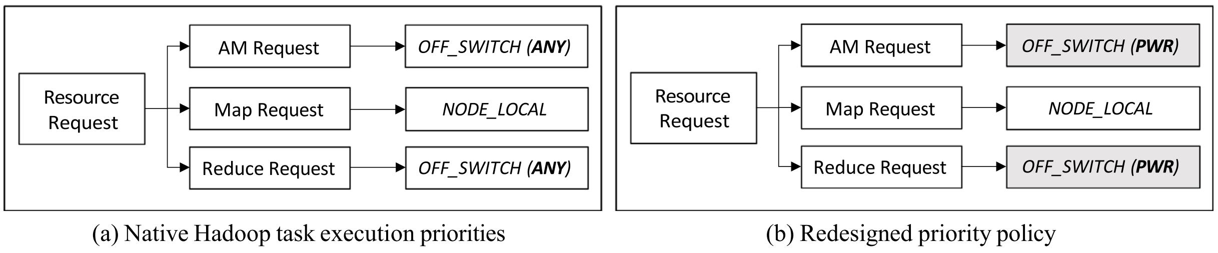


**Fig. 5.** Our redesigned ContainerAllocator component in the ResourceManager. The proposed policy assigns the ApplicationMaster(AM) and reduce tasks to powerful SBC nodes, not *any* SBC nodes in the heterogeneous SBC cluster.

specifications are sealed for each ResourceRequest. As in the figure, unlike map tasks, the lowest priority (OFF_SWITCH (ANY)) is assigned to both the AM and reduce tasks, which means they can be distributed to any cluster node. This implicit Hadoop policy causes substantial Hadoop performance variations on heterogeneous SBC Hadoop clusters.

To resolve this limitation, we designed a new data locality check policy for the AM and reduce tasks in Hadoop YARN's *ContainerAllocator* component by modifying the *OFF_SWITCH* policy (Fig. 5-(b)). The new policy assigns the AM and reduce tasks to powerful (PWR) SBC nodes, such as RPi 4B, not *any* SBC node. In other words, if a ResourceRequest belongs to either the AM or reduce tasks based on its priority, the *ContainerAllocator* component distributes them to powerful SBC nodes.

Even though both the AM and reduce tasks are assigned to powerful SBC nodes, their locations (e.g., same or separate nodes) also strongly influence the overall Hadoop performance. The AM and reduce tasks are bound to the cluster node that sent the earliest NODE_UPDATE heartbeat event to the RM and has available container slots. However, they may sometimes be assigned to an identical node due to a heartbeat transmission delay. Therefore, various distribution combinations can exist according to their locations on the cluster. For instance, all reduce tasks may be executed on the same node (worst case). The AM and the reduce task may be located on the same node. All the reduce tasks and AM are evenly assigned to separate nodes (best case). In this case, a node corresponds to a powerful (not weak) SBC node in the heterogeneous cluster. As the cluster size and the reduce task count increase, the complexity of the distribution combination noticeably increases. Thus, the native Hadoop inevitably show high performance variations according to locations of both the AM and reduce tasks. That is, the native Hadoop framework cannot guarantee the best performance for each job execution because it has no specific policy for the AM and reduce task assignments.

The AM and each reduce task should be evenly distributed to powerful SBC nodes for the best performance. Thus, we implemented a new operation in the *RMNode* component to manage the number of assigned reduce tasks on its node. When a map phase reaches a predefined overall progress (i.e., 5% by default), the redesigned *RMNode* component changes the state of the reduce container from SCHEDULED to RUNNING. This approach evenly distributes each reduce task to each powerful SBC node (not overlapped). Hence, the proposed design best employs cluster-level parallelism for the reduce task execution.

The RM requests an AM container based on the OFF_SWITCH level of the locality preference, causing the AM to be assigned to any node in the cluster. That is, the existing *ApplicationMaster* launcher supports a policy, where the AM is assigned to the node sending the prompt heartbeat for requesting the AM container execution. However, the proposed AM distribution policy waits for the heartbeat from one of the powerful nodes, and assigns the AM to the corresponding one. Without such considerations, it may assign the AM to one of the resource-frugal nodes which operate as a sequential processing mode. This causes a job execution time delay. Interestingly, unlike the reduce task distribution, we observed that the AM location has an almost negligible effect on the overall Hadoop performance regarding execution time and resource utilization. Thus, we do not provide any operation for the AM in the *RMNode* component.

### 3.3. Redesigned YARN architecture

This section consolidates all the proposed designs into the Hadoop YARN architecture and describes the overall processes.

#### 3.3.1. YARN service initialization

We proposed two frameworks for more effective task scheduling: master-driven and slave-driven frameworks. This section describes the overall processes of each design to set up the proposed MapReduce processing modes (i.e., parallel processing vs. sequential processing) during the Hadoop YARN service initialization. The proposed MapReduce processing modes are determined by the computational capability information that the *machineInfo* identifier for each SBC worker node provides.

The *machineInfo* identifier component is executed after each NM starts. It collects a local SBC node's computing resource information, such as the memory capacity and CPU specifications. Then, each framework determines an appropriate processing mode (i.e., parallel or sequential) as follows.

The master-driven design enables *NodeStatusUpdater* to communicate with RM to update NM's status via the heartbeat. Once the NM successfully starts, it sends node's *machineInfo* information to the RM. Once the RM accepts the NM registration request, it overrides the Hadoop configuration values by calculating the number of concurrently executable containers based on *machineInfo*. Finally, the master-driven framework chooses a parallel processing mode (for a powerful SBC node) or a sequential one (for a weak SBC node) accordingly.

On the other hand, the slave-driven design does not involve the RM in determining the processing mode. Instead, each SBC node in the cluster autonomously calculates the appropriate number of concurrent containers based on the *machineInfo* in each node. *ContainersLauncher* manages a thread pool for launching containers inside a node. This thread pool configures the maximum number of available threads. Based on this thread count, each SBC node selects an appropriate MapReduce processing mode (i.e., a local decision, not a global decision).

#### 3.3.2. After application submission

Creating the AM is the primary process for Hadoop YARN after submitting the Hadoop MapReduce application because MapReduce tasks depend on the AM component. Thus, the AM must be created by the RM before processing the MapReduce task. An AM execution mechanism is identical to a MapReduce task execution mechanism. Fig. 6 exhibits the redesigned overall YARN architecture. It also presents the overall processes of Hadoop YARN to create the AM component. For every application submitted to the RM, the ApplicationsManager initiates negotiations with the ResourceScheduler for AM container allocation. The AM container is the first container allocated; therefore, it is assigned to one of the powerful SBC nodes with available containers that sent the earliest NODE_UPDATE event to the RM component. After this initial setup, the *AMLauncher* component designates the corresponding node to initiate the AM process (i.e., JVM) for each submitted application.

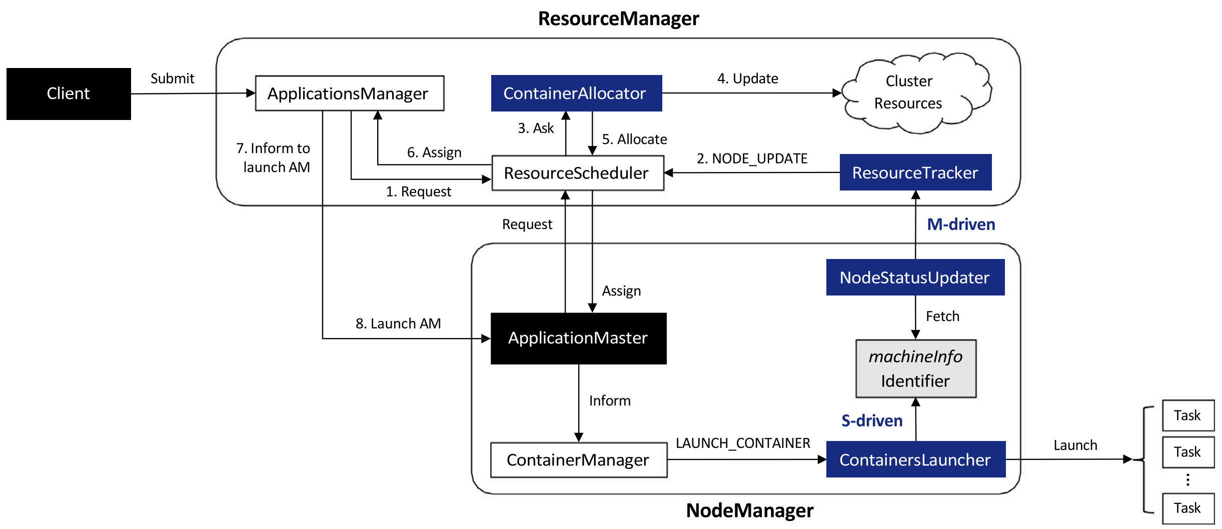


**Fig. 6.** Redesigned YARN architecture. Here, the component, Cluster Resources, represents the aggregation of each RMNode. Blue squares mark the redesigned components.

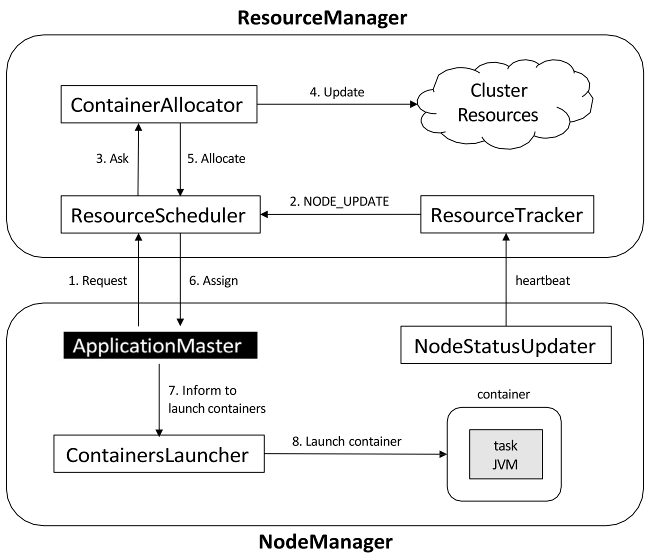


**Fig. 7.** The overall YARN MapReduce task scheduling process for launching each MapReduce container. Here, the Cluster Resources component represents the aggregation of each RMNode.

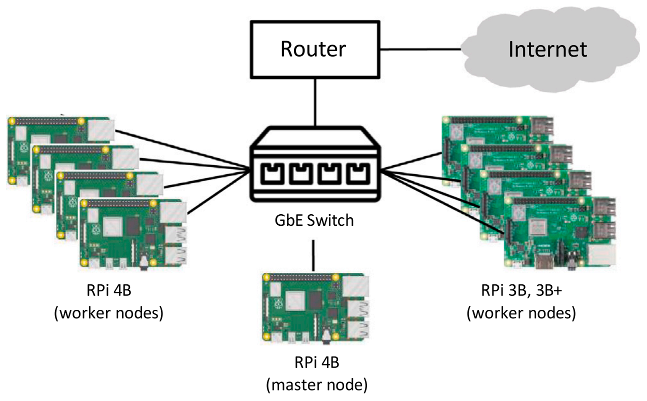


**Fig. 8.** Our heterogeneous SBC Hadoop cluster architecture. The SBC cluster consists of three types of Raspberry Pis, such as RPi 4B, 3B, and 3B+.

**Table 2**
SBC Hadoop cluster configurations.

| | Master node | Worker nodes | | |
|---|---|---|---|---|
| Node | RPi 4B (4 GB) | RPi 4B (4 GB) | RPi 3B+ | RPi 3B |
| The number of nodes | 1 | 4 | 1 | 3 |
| OS | Ubuntu 20.04.3 LTS (ARM 64-bit) | | | |
| Storage media | MicroSD 64 GB(Sandisk Extreme Pro) | | | |
| Framework | Apache Hadoop 3.2.3 | | | |
| Network switch | Gigabit Ethernet switch | | | |

Fig. 7 depicts the MapReduce task scheduling processes of Hadoop YARN after creating the AM. The AM requests containers from the RM using heartbeats. These periodic heartbeats contain a list of each container's locality information and task priority. On receiving this heartbeat, the RM waits for a heartbeat from an NM according to the locality preference. Once an NM is available for resource allocation, it reports its status to the RM. The *ResourceTracker* sends a NODE_UPDATE event to the *ResourceScheduler* component, enabling the *ResourceScheduler* to ask the *ContainerAllocator* to allocate resources. For the map task execution, the default *ContainerAllocator* assigns containers to map tasks according to the locality preference. Conversely, the redesigned *ContainerAllocator* is used for the reduce container assignment. It intelligently distributes each reduce task to a powerful SBC node that does not execute a reduce task. In the meantime, the AM periodically sends heartbeats to the RM to request resources. Once the AM receives containers, each NM launches the assigned containers.

## 4. Evaluation

This section provides the comprehensive experimental results and analyses.

### 4.1. Experimental setup

#### 4.1.1. Cluster configurations

Fig. 8 exhibits our heterogeneous SBC Hadoop cluster architecture. The Hadoop cluster comprises eight worker nodes (4×RPi 4B, 3×RPi 3B, and 1×RPi 3B+) and one master node (RPi 4B). They all connect to a gibabit Ethernet (GbE) network switch. Table 1 provides more detailed specifications for each RPi node.

Table 2 lists the cluster configurations. The Ubuntu 20.04 Long Term Support (LTS) 64-bit Operating System (OS) was installed instead

**Table 3**
Hadoop YARN configurations [20]. Here, all memory-related properties are in megabytes. The JVM heap memory size increases to 820 MB because the default value always causes an out-of-heap-memory error.

| **mapred-site.xml** | **Value (default)** |
|---|---|
| yarn.app.mapreduce.am.resource.mb | 1536 |
| yarn.app.reduce.am.resource.cpu-vcores | 1 |
| mapreduce.map(or reduce).memory.mb | 1024 |
| mapreduce.map(or reduce).cpu.vcores | 1 |
| mapreduce.map(or reduce).java.opts | -Xmx820m |
| **yarn-site.xml** | **Value (default)** |
| yarn.nodemanager.resource.memory-mb | 8192 |
| yarn.nodemanager.resource.cpu-vcores | 8 |
| yarn.scheduler.minimum-allocation-mb | 1024 |
| yarn.scheduler.maximum-allocation-mb | 4096 |
| yarn.scheduler.minimum-allocation-vcores | 1 |
| yarn.scheduler.maximum-allocation-vcores | 32 |
| yarn.nodemanager.vmem-pmem-ratio | 2.1 |

of the Raspberry Pi OS (previously called 'Raspbian') due to stability and compatibility problems [8]. To secure the RPi computing resources, we excluded a Graphic User Interface (GUI) installation, such as the GNOME desktop from Ubuntu 20.04 LTS. Moreover, to address a CPU thermal throttling problem of a powerful RPi 4B effectively [8], we applied an active cooling system with a fan and a passive one with heat sinks to all RPi nodes (including RPi 4B nodes). We employed default values of the Hadoop YARN configuration properties in Table 3 except the JVM heap memory size (*mapreduce.map(or reduce).java.opts*) because we found the default value always causes an out-of-heap-memory error. We also adopted default values of Hadoop configurations including 128MB of data chunks, 3 of the replication factor, the capacity scheduler for more objective evaluations.

There are no specific rules to appoint any node to a Namenode (i.e., master node) in the Hadoop cluster. However, since the Hadoop master node is responsible for heavy workloads including cluster resource management, job scheduling and monitoring, etc., it is common to manually assign a powerful node in the cluster to the Namenode when Hadoop clusters are configured. Therefore, we also designated a powerful SBC node (i.e., RPi 4B) for the separate Namenode in our heterogeneous SBC Hadoop cluster.

#### 4.1.2. Baseline

We found that SBC cluster nodes with a small RAM amount (such as third generation RPis with 1 GB of RAM) cannot run native Hadoop MapReduce applications with a large data set. That is, our heterogeneous RPi native Hadoop cluster cannot complete widely-known Hadoop benchmarks because RPi nodes with 1 GB of RAM always cause errors (i.e., node failure) due to insufficient memory space. To solve this limitation, we allocated an extra swap space (4 GB) to the third generation RPis. Thus, we adopted a native Hadoop framework with an extra swap space allocation as the baseline system.

#### 4.1.3. Performance metrics

We primarily measured the total elapsed time of Hadoop MapReduce applications. This total execution time consists of the Hadoop initial setup, map phase, reduce phase, and cleanup stage. This metric represents the actual Hadoop application performance. System resource usage: CPU utilization (sys, iowait), I/O traffic, and I/O requests were also measured.

For a fair evaluation, we adopted identical datasets and Hadoop configurations for the baseline framework and redesigned ones. Sync and drop cache operations were also performed for every iteration.

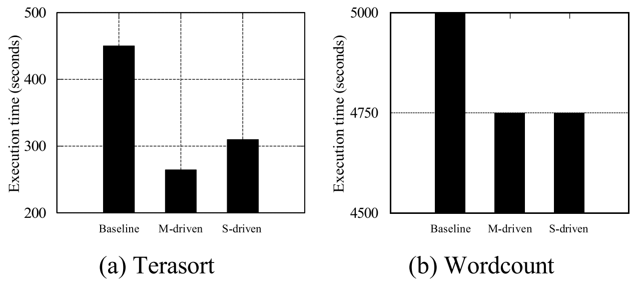


**Fig. 9.** Total execution time of native Hadoop (baseline) and our redesigned Hadoop (master-driven framework and slave-driven framework). Here, the best case performance is selected for native Hadoop. Our proposed design guarantees the better performance than the native Hadoop's best case performance. Here, a data size is 4 GB and a reduce task count is 4.

#### 4.1.4. MapReduce application benchmarks

We adopted two representative Hadoop MapReduce applications: Terasort as the I/O-intensive workload, Wordcount as the CPU-intensive workload. The Terasort benchmark sorts random data generated from the Teragen component. Terasort reads the input files, sorts them during a map phase, and writes HDFS output files during the reduce phase. Unlike Wordcount, which typically generates the final data of a reduced size, Terasort does not change its data size. It simply rearranges (i.e., sorts) the data sequence in ascending or descending order.

Wordcount benchmark counts the number of separate word occurrences from input data (i.e., text or a sequence file). A map function splits input data into each word (regarding the key–value pair) and generates intermediate data corresponding to a reduce function's input data. Finally, the reduce function aggregates each map's intermediate data to the final word count data.

The main goal of this work is not to run various Hadoop applications, but to verify our redesigned Hadoop frameworks on heterogeneous SBC clusters to explore their possibilities and challenges as pioneering work. Both Wordcount and Terasort applications are widely employed as representative Hadoop MapReduce benchmark applications because they have different workload characteristics (I/O-intensive vs. CPU-intensive workloads).

### 4.2. Experimental results

This section presents various experimental results and comparative analyses.

#### 4.2.1. Overall performance

Table 4 presents the overall performance (i.e., total execution time) of the original Hadoop framework (baseline) and the redesigned frameworks (master-driven and slave-driven). The AM and all reduce tasks can be distributed to any SBC node in the entire cluster because the original Hadoop framework has no specific rules for their assignment to cluster nodes. Thus, we can observe various task assignment combinations according to their distributions on each cluster node. Intuitively, the baseline's best performance (i.e., referred to as the best) occurs when ApplicationMaster(AM) and all reduce tasks are evenly distributed to powerful RPi 4B nodes. Similarly, the native Hadoop's worst performance (i.e., referred to as the worst) is observed when AM and all reduce tasks are assigned to weak SBC nodes (i.e., RPi 3Bs). As a result, for each execution of an identical MapReduce application, the native Hadoop performance varies from *worst* to *best* in Table 4. For experiments, we manually assigned both tasks to all weak nodes for the worst-case baseline experiment. Similarly, both were manually distributed to all powerful nodes for the best-case one.

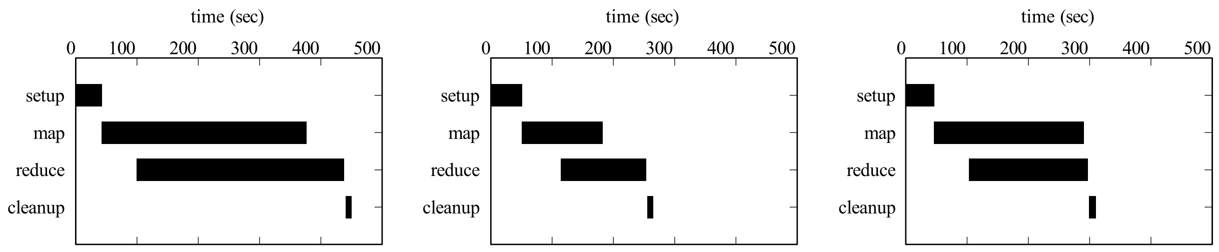
**Table 4**
Total execution time (seconds). Redesigned Hadoop frameworks (master-driven and slave-driven) outperform a native Hadoop framework (baseline). The data chunk size is 128 MB.

| # of reduces | Input | Native Hadoop | | Master-driven | Slave-driven |
|---|---|---|---|---|---|
| | data size | worst | best | | |
| 1 | 1 GB | 500.7 | 209.1 | 158.6 | 196.8 |
| | 2 GB | 1167.5 | 403.7 | 252.9 | 354.7 |
| | 4 GB | 2745.0 | 718.4 | 692.5 | 629.8 |
| | 8 GB | 5254.5 | 1453.3 | 1340.0 | 1284.0 |
| 2 | 1 GB | 461.3 | 143.3 | 142.7 | 143.6 |
| | 2 GB | 957.6 | 287.5 | 176.0 | 241.6 |
| | 4 GB | 1599.9 | 526.5 | 446.0 | 455.4 |
| | 8 GB | 2889.8 | 1004.1 | 637.3 | 736.1 |
| 4 | 1 GB | 243.6 | 190.7 | 141.9 | 147.8 |
| | 2 GB | 475.9 | 280.1 | 152.5 | 216.2 |
| | 4 GB | 928.9 | 450.2 | 264.4 | 309.7 |
| | 8 GB | 1701.9 | 862.1 | 506.3 | 670.3 |

(a) Terasort

| # of reduces | Input | Native Hadoop | | Master-driven | Slave-driven |
|---|---|---|---|---|---|
| | data size | worst | best | | |
| 1 | 1 GB | 2468.7 | 1413.9 | 1413.7 | 1431.8 |
| | 2 GB | 5486.7 | 3025.8 | 2823.8 | 2862.6 |
| | 4 GB | 14 922.8 | 6902.3 | 6830.2 | 6690.3 |
| | 8 GB | | | – | |
| 2 | 1 GB | 1973.3 | 1330.6 | 1205.1 | 1337.5 |
| | 2 GB | 5772.2 | 2785.2 | 2493.6 | 2419.7 |
| | 4 GB | 12 419.4 | 5977.1 | 5798.7 | 5691.9 |
| | 8 GB | | | – | |
| 4 | 1 GB | 2522.0 | 1173.1 | 1204.5 | 1213.9 |
| | 2 GB | 4627.4 | 2471.1 | 2100.4 | 2298.2 |
| | 4 GB | 10 268.0 | 4990.7 | 4740.8 | 4742.9 |
| | 8 GB | – | – | 10 492.7 | 9958.3 |

(b) Wordcount



(a) Native Hadoop (best case) (b) Master-driven (c) Slave-driven

**Fig. 10.** Total execution time breakdown of MapReduce tasks. Here, each framework adopts 4 GB data and 4 reduce tasks under Terasort. Here, the reduce phase represents the combined time of both the shuffle phase and the reduce phase.

The proposed design performs better than the native Hadoop by an average of 2.55× (master-driven) and 2.19× (slave-driven) under Terasort, and 1.55× (master-driven) and 1.52× (slave-driven) under Wordcount. More importantly, the proposed design guarantees better performance than the best case performance of the native Hadoop framework. Fig. 9 exhibits the total execution time for both the baseline and redesigned Hadoop frameworks (master-driven and slave-driven). For a very conservative evaluation, we compared the redesigned Hadoop performance with the best performance of the native Hadoop. As in Fig. 9, the redesigned Hadoop achieves higher performance than the original Hadoop regarding Terasort and Wordcount due to the effective task scheduling policies of the redesigned Hadoop YARN.

For a deeper analysis, Fig. 10 breaks down the total execution time of each framework into four Hadoop execution phases (setup, map, reduce, and cleanup). Here, the reduce phase represents the combined time of both the shuffle phase (the summation of the copy and sort stage) and the reduce phase. It adopts 4 GB of data and four reduce tasks under Terasort (Fig. 9-(a)). For the setup phase, each Hadoop framework initializes the AM and prepares each requested container. Unlike the native Hadoop, the proposed design carefully assigns the AM to one of the powerful RPi nodes (i.e., RPi 4B) based on the *machineInfo* information each worker node provides. Although these processes require extra setup time, it is a negligible one-time process (about a few milliseconds). After the MapReduce job is completed, a cleanup task is executed. We did not change this cleanup logic; thus, each Hadoop framework has an identical cleanup time.

Unlike the setup and cleanup times, the map and reduce phases have significantly different execution times between the baseline and proposed frameworks. For more effective map task processing, we proposed master-driven and slave-driven frameworks. Moreover, new AM and reduce task distribution policies are proposed to more intelligently assign the reduce tasks (and AM) to heterogeneous SBC cluster nodes. Consequently, both proposed designs exhibit an average of 2.53× and 1.37× faster map task execution time than the map task execution time (best case) for the baseline. In addition, they also achieve 2.44× and 1.75× faster reduce execution time than that of the baseline (best case).

**Table 5**
MapReduce job performance of each Hadoop framework (baseline vs. the proposed design) on a single RPi 3B worker node (seconds).

| Benchmarks | Chunk sizes | # of chunks | Baseline | M-driven | S-driven |
|---|---|---|---|---|---|
| Terasort | 64 MB | 2 | 166.4 | 172.2 | 173.1 |
| | | 4 | 359.7 | 269.7 | 280.3 |
| | | 8 | 911.4 | 462.6 | 494.9 |
| | 128 MB | 2 | 274.4 | 272.2 | 311.1 |
| | | 4 | 575.2 | 446.8 | 466.6 |
| | | 8 | 1267.9 | 854.4 | 905.8 |
| Wordcount | 64 MB | 2 | 749.2 | 1157.3 | 1140.8 |
| | | 4 | 1372.5 | 2226.4 | 2218.1 |
| | | 8 | 4397.6 | 4515.3 | 4579.2 |
| | 128 MB | 2 | 1538.1 | 2490.5 | 2412.4 |
| | | 4 | 2466.3 | 4838.0 | 4857.6 |
| | | 8 | 6729.8 | – | – |

Please note that processing more than 8 GB of data under Wordcount raises another challenging problem in our SBC cluster configurations. This limitation results from insufficient storage space. Section 5.2 discusses this problem in more detail.

### 4.2.2. Design verification

Due to insufficient memory, the original Hadoop framework cannot run MapReduce tasks with a large dataset on resource-frugal SBC nodes (i.e., RPi 3B and 3B+ with 1 GB of RAM). It attempts to assign excessive MapReduce tasks to them without carefully considering their computing resources, which eventually causes node failures.

To resolve this, we added an extra swap space into the third generation RPi Hadoop nodes. However, we redesigned the Hadoop YARN architecture to consider nodes' heterogeneity judiciously in the SBC Hadoop cluster. The proposed design effectively uses particularly such SBC nodes with low computing resources. To verify the effectiveness of the design, we built a single resource-frugal worker node (RPi 3B with 1 GB of RAM) cluster with a master node (RPi 4B with 4 GB of RAM).

Table 5 presents the overall performance for each Hadoop framework (i.e., the baseline vs. our design) on a single RPi 3B worker

**Table 6**
Native Hadoop performance for each combination of AM and two reduce task locations on a heterogeneous RPi Hadoop cluster (seconds). Each column title of the second row corresponds to two reduce task locations. All (AM and two reduce tasks) tasks on RPi 3Bs [4Bs] correspond to the worst [best] performance case. Here, two reduce tasks are assigned to each node.

| Chunk size | Input | AM on RPi 3B | | | AM on RPi 4B | | |
|---|---|---|---|---|---|---|---|
| | data size | 3Bs | 3B&4B | 4Bs | 3Bs | 3B&4B | 4Bs |
| 64 MB | 1 GB | 441 | 433 | 282 | 460 | 263 | 219 |
| | 2 GB | 998 | 717 | 410 | 739 | 478 | 379 |
| | 4 GB | 1857 | 1756 | 745 | 1497 | 1223 | 683 |
| | 8 GB | 3300 | 3206 | 1177 | 3019 | 2278 | 882 |
| 128 MB | 1 GB | 461 | 310 | 186 | 453 | 321 | 143 |
| | 2 GB | 958 | 686 | 255 | 827 | 618 | 287 |
| | 4 GB | 1600 | 1540 | 603 | 1242 | 1300 | 527 |
| | 8 GB | 2890 | 2846 | 1059 | 2685 | 2541 | 1004 |

(a) Terasort

| Chunk size | Input | AM on RPi 3B | | | AM on RPi 4B | | |
|---|---|---|---|---|---|---|---|
| | data size | 3Bs | 3B&4B | 4Bs | 3Bs | 3B&4B | 4Bs |
| 64 MB | 1 GB | 2891 | 1475 | 1300 | 2423 | 1267 | 1267 |
| | 2 GB | 5992 | 3039 | 2923 | 5386 | 2730 | 2709 |
| | 4 GB | 12 437 | 12 336 | 6503 | 11 592 | 11 991 | 6024 |
| | 8 GB | | – | | | – | |
| 128 MB | 1 GB | 1973 | 1543 | 1293 | 2033 | 1233 | 1331 |
| | 2 GB | 5772 | 3016 | 2812 | 5150 | 2807 | 2785 |
| | 4 GB | 12 419 | 6046 | 5980 | 12 094 | 6042 | 5977 |
| | 8 GB | | – | | | – | |

(b) Wordcount

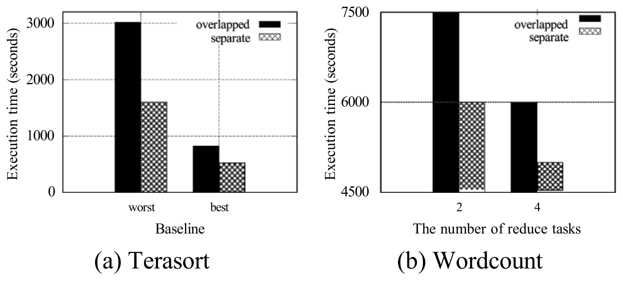


**Fig. 11.** Native Hadoop performance for the best (all AM and reduce tasks on RPi 4B nodes) and worst (all AM and reduce tasks on RPi 3B nodes) cases. Each case is classified into two additional cases (separate vs. overlapped) in terms of the reduce task locations. Here, the dataset size is 4 GB and the reduce task count is two (Figure (a)) or four. Figure (b) corresponds to the best performance case.

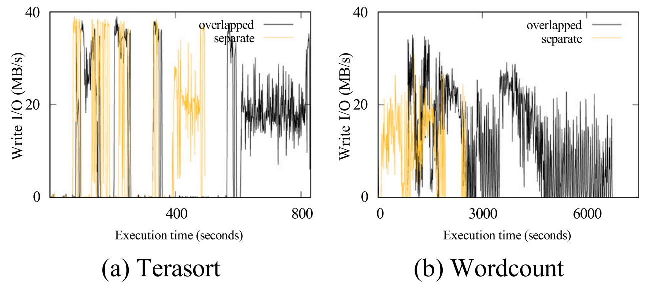


**Fig. 12.** Write I/O traffic over time during the reduce phase of each application with a 4 GB dataset and two reduce tasks.

node. The two proposed designs (master-driven and slave-driven) display no noticeable performance difference because this experiment is performed on a single worker node cluster. Their MapReduce task schedulers should assign all tasks to the single worker node (RPi 3B node). The proposed frameworks have linear scalability as the data size increases because the proposed Hadoop frameworks sequentially process all tasks particularly on the weak SBC node to prevent a node failure. However, we observed a frequent system hang (i.e., freeze) when they processed a large dataset (here, eight chunks). This study found that this problem originates from an OS page cache amount when the chunk count exceeds four. This excessive page cache eventually leads to memory pressure. To avoid this limitation, we periodically dropped the page cache.

On the other hand, the total execution time for the baseline significantly (more than linearly) increases as the data chunk count increases because the baseline's swapping operations inevitably raise significant extra I/O overhead as the dataset size increases. Thus, the baseline tends to favor CPU-intensive workloads (Wordcount) over I/O-intensive workloads (Terasort). However, our designs catch up with the original Hadoop performance under Wordcount as the data chunk count increases due to the linear scalability of the proposed design.

### *4.2.3. Reduce task distribution*

A native Hadoop, unlike map task assignment, does not have specific policies for assigning the AM and reduce tasks to cluster nodes. Therefore, noticeable performance variations occur according to their locations on the cluster nodes. This section explores the correlation between performance and task (reduce tasks and AM) locations on a heterogeneous RPi Hadoop cluster.

Several combinations exist for AM and reduce task locations. For instance, given two reduce tasks, AM can be assigned to an RPi 3B node, and two reduce tasks can be executed on either a single RPi 3B node together ('overlapped' reduce tasks) or two respective RPi 3B nodes ('separated' reduce tasks). This performance is the worst case example. Similarly, AM can be located on a RPi 4B and two reduce tasks can be assigned to either a single RPi 4B node ('overlapped' reduce tasks) or two individual RPi 4B nodes ('separated' reduce tasks), which is the best case. In addition to these cases, several more location combinations are possible.

Table 6 presents the native Hadoop performance for each combination of the AM and two reduce task locations on the heterogeneous RPi Hadoop cluster. This table assumes each reduce task is allocated to individual node (not the same node) because this separate reduce task execution performs better than an overlapped reduce task configuration where each reduce task is executed on the same node (i.e., single node). The experiments in Table 6 demonstrate that the reduce task distribution is a more critical factor than the AM location concerning Hadoop MapReduce performance. The AM location has a negligible effect on the total execution time and resource utilization. Thus, this section focuses more on the reduce task distribution.

As in Table 6-(b), the baseline fails to complete Wordcount with the 8 GB dataset because of the out-of-heap-space error during the reduce phase. Unlike Terasort, Wordcount consumes considerable OS page cache space to store the intermediate data of map tasks temporarily. Furthermore, the reduce tasks require an additional memory space on top of the map phase, eventually causing the Wordcount job failure. The Hadoop framework is based on JDK 1.8, which allows the JVM heap memory to be created only in Dynamic Random Access Memory (DRAM), not in any alternative devices, such as Solid State Drives (SSDs) or Hard Disk Drives (HDDs). Thus, the native Hadoop framework with an extra swap space fundamentally cannot resolve this limitation.

Fig. 11-(a) exhibits the native Hadoop performance for the best and worst performance cases. Each case is classified into additional two cases, separate *vs.* overlapped, in terms of the locations of the two reduce tasks. In the worst case (i.e., all AM and reduce tasks on RPi 3Bs), the separate reduce task assignment on each RPi 3B node has 1.88× better performance than the overlapped reduce task assignment on a single RPi 3B node. Similarly, when all AM and reduce tasks run on RPi 4Bs (i.e., the best case), the 'separate' approach has a 1.73× faster execution time than the 'overlapped' approach. Wordcount also exhibits

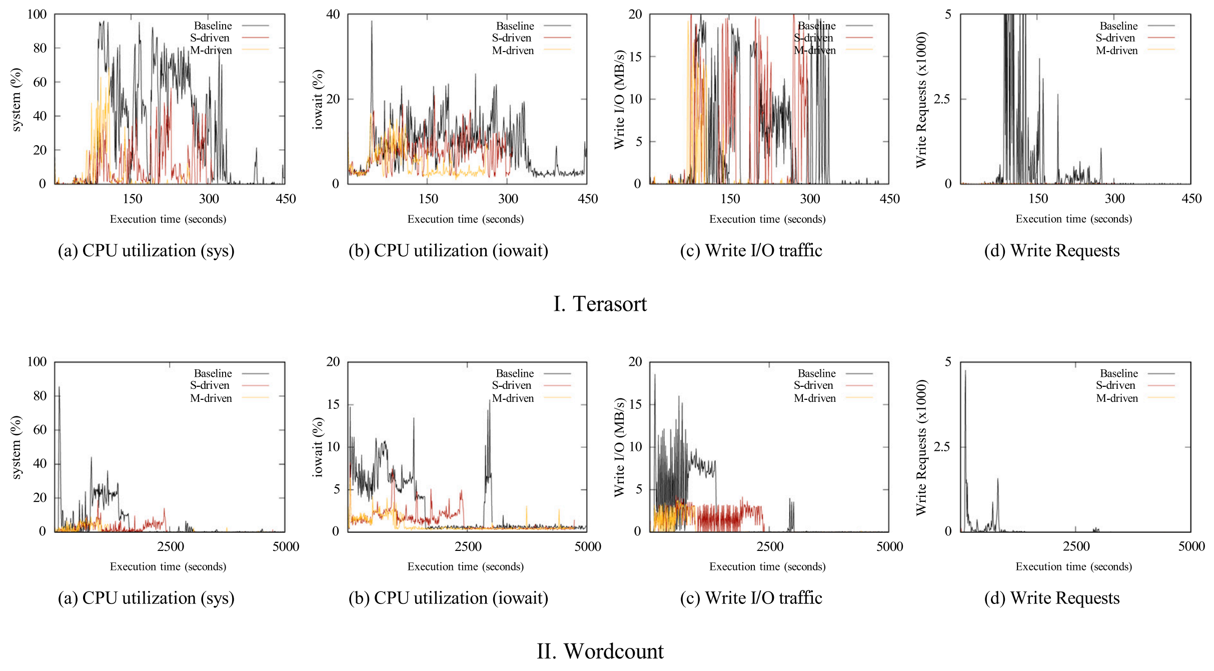


**Fig. 13.** Various resource utilization types on RPi 3B nodes. The proposed Hadoop frameworks consume less CPU resources and generate less I/O traffic.

a pattern similar to Terasort; therefore, we omitted it. The reduce task count as well as the reduce task location also has a noticeable effect on the overall Hadoop MapReduce performance. As in Fig. 11-(b), when it increases from two to four, the baseline performance also increases by an average of 1.23×.

Fig. 12 captures the write I/O traffic over time during the reduce phase of each application. Reduce nodes fetch the output data of each map task over the Hyper Text Transfer Protocol (HTTP) and write them to their local disks [40,41]. Thus, overlapped reducers write each fetched data to a single reduce node, generating higher write traffic in the node. 'Overlapped' produces 1.06× and 2.49× more write traffic in the reduce node under Terasort and Wordcount, respectively. This different traffic results from different workload characteristics during the reduce phase of each application, where Wordcount creates very heavy intermediate data for map tasks.

These comprehensive experimental results inspired our redesigned Hadoop YARN scheduler (i.e., AM and reduce task distribution policies).

### 4.2.4. *Resource utilization*

This section explores the performance of the redesigned Hadoop framework concerning the swap-in/out operations. Admittedly, while Terasort is a representative I/O-intensive application, Wordcount is a CPU-intensive application [42]. The sorting process involves frequent data transfers between the RAM and external storage. The native Hadoop framework inevitably triggers frequent swapping operations under I/O bound workloads, decreasing performance. To verify this, we measured the baseline's swap-in/out operation on the RPi 3B node by using the Linux *vmstat* command while running both Terasort and Wordcount.

In the worst case environment (i.e., all reduce tasks and AM on RPi 3B nodes), Terasort generates an average of 2.2× and 1.3× more swap-in and swap-out operations, respectively than in the best case environment (i.e., all reduce tasks and AM on RPi 4B nodes). Excessive swap operations unavoidably cause extra I/O overhead for running applications. However, Wordcount produces a few I/O operations, causing a low number of swap-in/out operations. In the best case environment, Wordcount creates an average of 2.61× and 4.37× fewer swap-in/out operations, respectively than Terasort. These experiments imply that the proposed Hadoop framework benefits more from I/O-intensive applications due to the effectively redesigned Hadoop YARN architecture, which does not trigger extra I/O traffic. Fig. 13 illustrates the various resource utilization types on RPi 3B nodes for each framework (i.e., the baseline and two proposed frameworks).

**CPU utilization**: Swap operations cause system-level interrupts and context switches, directly affecting CPU utilization. Our proposed frameworks consume an average of 3.28× and 3.5× less CPU than the baseline Hadoop framework for Terasort and Wordcount, respectively. We also investigated *iowait*, which indicates the percentage of time when the CPU is idle and waiting for I/O operations to complete. Frequent swap operations lead to a high *iowait* since the CPU spends more time waiting for data to be read from or written to the swap space. As in Fig. 13-(b), our frameworks exhibit a lower *iowait* than the baseline by an average of 1.27× and 1.80× under Terasort and Wordcount, respectively.

**I/O traffic**: Swapping involves reading data from memory and writing them to disks (swap space). This method generates significant I/O traffic, creating a performance bottleneck, especially on systems with slow disks or heavy I/O loads. Fig. 13-(c) presents the write I/O traffic for each Hadoop framework. Our frameworks generate an average of 1.67× and 2.12× less write I/O traffic under Terasort and Wordcount. For the read I/O, the proposed frameworks produce 1.26× and 2.08× less traffic, respectively. We omitted this read I/O plot due to its similarity to the write I/O patterns.

**I/O requests**: The I/O requests represent the number of merged requests in the device queue per second. When I/O requests stay in a device queue, they wait for their turn until the device executes them. Thus, swapping has a significant influence on I/O requests. The proposed frameworks outperform the native Hadoop by an average of 13.86× and 32.98× fewer write I/O requests and 73.24× and 39.28× fewer read I/O requests under Terasort and Wordcount, respectively (Fig. 13-(d)).

### 4.2.5. *Master-driven vs. Slave-driven*

This section discusses the proposed frameworks (master-driven and slave-driven). Table 4 indicates that our redesigned Hadoop frameworks always perform better than the *best* (not the average) performance of the native Hadoop framework. For instance, the master-driven design achieves an average performance of 1.41× and 1.24×

**Table 7**
Average map task distribution rates for each Hadoop framework. It represents a ratio of the successfully executed map task count on RPi 3B nodes to the total map task count on the RPi Hadoop cluster.

| Benchmark | Input data size | Native Hadoop | | Master-driven | Slave-driven |
|---|---|---|---|---|---|
| | | worst | best | | |
| Terasort | 2 GB | 0.60 | 0.52 | 0.25 | 0.56 |
| | 4 GB | 0.30 | 0.35 | 0.13 | 0.35 |
| | 8 GB | 0.09 | 0.11 | 0.20 | 0.40 |
| Wordcount | 2 GB | 0.17 | 0.21 | 0.06 | 0.19 |
| | 4 GB | 0.09 | 0.19 | 0.13 | 0.19 |
| | 8 GB | – | – | 0.06 | 0.19 |

better than the best baseline performance under Terasort and Wordcount, respectively. More interestingly, the master-driven framework exhibits slightly higher performance than the slave-driven framework: on average, 1.16× and 1.018×, respectively for Terasort and Wordcount. This outcome is because they provide different task distribution policies according to each node's computing resource information (i.e., *machineInfo*).

Table 7 presents the average map task distribution rates for each Hadoop framework. Specifically, it represents a ratio of a successfully executed map task count on RPi 3B nodes (i.e., resource-frugal SBC nodes) to the total map task count in the cluster. As the input data size increases, this ratio of the native Hadoop decreases. Unlike the proposed frameworks, the baseline concurrently processes multiple map tasks on each RPi 3B node with the help of the extra 4 GB swap space. More extensive input data inevitably trigger more frequent swapping I/O operations on weak worker nodes, which reduces the chance of subsequent map task assignment to the overburdened RPi 3B nodes.

The redesigned Hadoop frameworks present a different map task distribution on RPi 3B nodes. Task scheduling policies of the master-driven framework concentrate more on a faster total execution time because the global RM effectively assigns each task to each worker node based on the computing resources. Thus, the master-driven design tends to distribute as many tasks as possible to the powerful worker nodes, such as RPi 4B, resulting in relatively low map task distribution rates on RPi 3B. Contrarily, the slave-driven design somewhat considers task load balancing as well as the total execution time because each local NM on the respective worker nodes determines their appropriate task scheduling mode based on their computing resources. As in Table 7, the slave-driven framework has higher map task distribution rates on RPi 3B nodes than the master-driven framework.

This task distribution rate is closely associated with a task wave that is the maximum number of concurrently executable containers in the cluster. This task wave corresponds to a total summation of the concurrent container count for each worker node in the cluster. The number of the concurrent containers, $CC$, is directly calculated from the Hadoop YARN configurations for each worker node (Table 3) using the following equation:

$$CC = \frac{yarn.nodemanager.resource.memory - mb}{mapreduce.map(or\ reduce).memory.mb} \tag{1}$$

Based on Eq. (1), the RM in our Hadoop cluster assigns eight concurrent containers for each RPi worker node; thus, the cluster's task wave corresponds to 64 (i.e., 8 concurrent containers × 8 worker nodes). However, unlike the slave-driven design, the master-driven framework's task wave in our cluster is 36 (not 64): 8 concurrent containers × 4 RPi 4B worker nodes and 1 container × 4 third generation RPi worker nodes.

In the master-driven design, the RM (not the NM) controls the concurrent container count of each worker node and changes the count from eight to one for the resource-frugal RPi 3B (or 3B+) nodes based on their computing resource information. For example, assuming the input data size is 8 GB and the chunk size is 128 MB, 64 total data chunks exist in the cluster. Although our slave-driven design has a wave factor of one, the master-driven design has a wave factor of two. The wave factor is calculated by dividing the total data chunk count by the cluster's task wave. Therefore, our slave-driven framework can process all map tasks in a single wave because its wave factor is one. However, the proposed master-driven design requires two waves to process them. Thus, for each wave, the RM of the master-driven framework attempts to assign as many map tasks as possible to the powerful SBC worker nodes (RPi 4B) while the weak RPi 3B or 3B+ nodes are still processing their map tasks. This is the main reason the master-driven framework somewhat favors the total execution time reduction. Moreover, to assign a map task to a performance-frugal SBC node, the RM of the master-driven framework first creates a container in a weak worker node and runs a map task in the container. Then, it withdraws the container once the map task is completed on the weak node. The master-driven design requires these three steps for each map task execution on the performance-frugal SBC node, causing extra overhead.

Conversely, the proposed slave-driven framework utilizes a thread pool to efficiently launch each container. Even though the NM of the slave-driven design sets the maximum thread count to one to prevent node failure (i.e., for sequential map task processing like the master-driven design), its scheduling overhead is lower than the master-driven framework. Thus, relatively more map tasks can be effectively executed on the performance-frugal SBC nodes. Consequently, the slave-driven design exhibits higher average map task distribution rates on RPi 3B nodes than the master-driven design.

More interestingly, all three designs exhibit noticeably lower map task distribution rates on RPi 3B nodes under Wordcount than Terasort due to more frequent Hadoop speculative executions. While a single map task processing on the RPi 3B node takes an average of 56 s for Terasort, it takes significantly longer, on average 906 s for Wordcount. This long map phase on RPi 3B nodes under Wordcount more frequently triggers speculative execution on other nodes in the cluster, causing lower map task distribution rates on RPi 3B nodes.

## 5. Discussion

This section discusses other interesting problems regarding our heterogeneous SBC cluster study.

### 5.1. Uberized jobs

Uber mode in Hadoop is a mode that allows MapReduce tasks to run through the AM's JVM without allocating a separate container for each task to save memory [43]. That is, multiple containers can be launched on the same JVM, enabling the execution of multiple tasks simultaneously on a resource-frugal SBC worker node without creating individual task JVMs (i.e., YarnChild). We observed that, unlike the experimental results in Fig. 2, two map tasks can concurrently process 128 MB data under this uber mode without causing the OOM problem. The Hadoop uber mode reduces communication overhead within JVMs as well as container initialization and termination overhead. As a result, two concurrent map tasks can be successfully executable on such a resource-frugal RPi 3B node.

However, the uber mode in Hadoop has the following strict limitations. It supports (1) map-only jobs with at most nine tasks, or (2) jobs with a single reducer, or (3) the input data size smaller than the HDFS chunk size. Due to these three constraints, this mode is inappropriate for typical Hadoop cluster environments.

### 5.2. Temporary data

The volume of Hadoop temporary data significantly influences the SBC Hadoop cluster because SBCs generally adopt limited storage

**Table 8**
All temporary data for processing the 8 GB dataset (i.e., 64 × 128 MB data chunks) with two reduce tasks on the cluster of eight RPi worker nodes, measured on a reduce node (RPi 4B) in the slave-driven framework. All map output data are assumed to be fetched evenly (i.e., eight map tasks for each worker node), and all map tasks are completed successfully.

| Temporary data | Terasort (MB) | Wordcount (MB) |
|---|---|---|
| (a) Spilled data in the map phase | – | – |
| (b) Intermediate output data after the map phase | 134 × 8 | 653 × 8 |
| (c) Fetched intermediate data | – | 327 × 64 |
| (d) Merged data in the reduce phase | 400 × 10 | 1500 × 10 |

devices, such as micro Secure Disks (microSDs) or embedded Multi-Media Controller (eMMC) cards [44]. During MapReduce processing, the Hadoop framework generates four kinds of temporary data: (1) spilled data in the map phase, (2) intermediate output data after the map phase, (3) fetched intermediate data, and (4) merged data in the reduce phase.

Each map task has a circular memory buffer (100 MB by default). The data are spilled to the disk once the buffer reaches a threshold (80% by default). Given that the HDFS block size is 128 MB, while the Hadoop framework generates the same spill data amount as the chunk size under Terasort, Wordcount produces 20 spilled data files each of which is 31 MB. After completing a map task, these spilled data are immediately deleted. However, if a map task fails, the corresponding spilled data remain until the job successfully ends.

Unlike the spilled data, map output data are not promptly removed from the disk once any reducer fetches them in case of the reducer's failure. Therefore, they should stay on disk until the job succeeds. For each map task, we observed both Terasort and Wordcount generated 134 MB (128 MB plus meta data) and 653 MB (31 MB × 20 plus meta data) of intermediate map output data, respectively.

Each reduce task copies a corresponding partition of each map output data from the cluster worker nodes over HTTP by using a buffer (300 MB by default). Terasort does not write the fetched data onto disk because they are small enough to be processed entirely in the buffer. However, Wordcount spills considerable fetched data onto the disk depending on the input data.

Finally, all partitions fetched from each map output data across the cluster must be merged before reduce task processing. This merge phase consists of multiple rounds, depending on map output counts and a merge factor (10 by default) [20]. As in Table 8, Terasort and Wordcount produced 400 MB and 1.5 GB of merged data for each merge round.

Table 8 lists the total amount of all temporary data on a single RPi 4B node hosting one reduce task in the proposed slave-driven Hadoop framework. Input data are 8 GB, and the reduce task count is two. As all map tasks are completed, (a) the spilled data in the map phase do not remain during MapReduce processing.

Whereas Terasort produces about 5 GB of all temporary data on a single RPi node with the reduce task, Wordcount generates about 35 GB of all temporary data. Although they do not exceed our storage limit (64 GB) for each RPi node, their size is never negligible considering the resource-frugal SBC cluster environment. Moreover, the native Hadoop framework requires an extra swap space (4 GB). In practice, map output partitions are not evenly generated, which may cause an out-of-storage-space error on a worker node. Such a worker node is shut down and eventually reaches an inactive state. Therefore, we observed that particularly Wordcount, unlike Terasort, cannot complete with 8 GB of input data (Table 4). When this Wordcount with 8 GB of input data is executed on a single desktop PC, temporary data reach about 83 GB.

To secure the storage space in the resource-frugal SBC nodes, we may consider reducing the HDFS replication factor to one or map output data compression. However, the low replication factor may achieve storage savings at the cost of the Hadoop fault tolerance. Map output compression significantly lengthens the total execution time. Adopting external storage devices via USB 3.0 ports may be an additional option. However, many earlier SBC generations still employ low-speed I/O ports, such as USB 2.0 ports.

### 5.3. Different performance views

Lee et al. [8] demonstrated that a Hadoop cluster of five RPi 4B worker nodes performed better than a single desktop PC in terms of performance-per-watt and per-dollar. In addition, their study implied that a cluster of seven RPi 4B nodes could provide an overall Hadoop performance comparable to a single desktop PC.

This section discusses the possibilities of the proposed framework (i.e., master-driven) on the heterogeneous Hadoop cluster (for configurations, see Table 2). To validate this, we installed the proposed master-driven Hadoop framework on a single desktop PC (i5-9600K@3.7 GHz (six cores), 8 GB of RAM, 500 GB of SSD). We measured the total execution time and power consumption under the Terasort benchmark with 32 GB of input data. The single desktop PC exhibits a 2.73× faster execution time than the heterogeneous RPi cluster. The PC costs about $600 (USD), and the heterogeneous cluster costs about $635 (i.e., $35 for RPi 3B and 3B+, $55 for RPi 4B 4 GB RAM, $10 for each storage media, and $130 for the network setup), implying that the proposed Hadoop cluster still does not exhibit performance-per-dollar comparable to the single PC. In addition, the PC consumes about 102 W and the cluster requires approximately 55 W during the benchmarking. Though the heterogeneous RPi cluster exhibits 1.85× lower power consumption, it does not display performance comparable to the PC per-dollar or per-watt, primarily because the previous generation RPi nodes never matched the desktop PC for either metric. Their overall performance is significantly lower than the latest generation RPi 4B nodes even though their unit price is identical or similar to RPi 4B [8]. Therefore, we can expect that heterogeneous RPi clusters consisting of the latest RPi 4B or future generation RPi nodes may exhibit promising possibilities for big data processing in terms of the performance metrics.

### 5.4. Runtime overhead

Our redesigned YARN scheduler's runtime overhead is negligible compared to the original one. Most parts of the proposed scheduler algorithms follow the original YARN's procedures except the following main parts;

First, during the YARN service initialization, the proposed design determines a MapReduce processing mode (i.e., either a parallel processing mode or a sequential processing mode) based on the computational capabilities that the machineInfo identifier component for each SBC worker node provides. This is just a one-time (not periodic) communication process between the NM and the RM after each NM starts. Thus, this runtime cost is negligible.

Second, we redesigned data locality check policies for both the AM and the reduce tasks distribution in Hadoop YARN's ContainerAllocator component by modifying the OFF_SWITCH policy. If a ResourceRequest belongs to either the AM or reduce tasks based on its priority, the ContainerAllocator component distributes them to powerful SBC nodes. To implement this policy, we simply added one nested *if-statement* in the ContainerAllocator component to check container priority and to assign the corresponding (AM or reduce) task to one of the powerful SBC nodes (i.e., RPi 4B node). Unlike map task sending a frequent ResourceRequest, they do not transmit it frequently. That is, they are allocated to powerful cluster nodes only once for each application submission. Therefore, this runtime cost is also very negligible.

## 6. Conclusion

This paper presents the redesigned Hadoop YARN architecture to improve Hadoop MapReduce application performance on heterogeneous SBC Hadoop clusters, proposing two Hadoop framework designs: master-driven and slave-driven frameworks. Both provide different MapReduce task scheduling mechanisms based on the computing resource information from each heterogeneous SBC node in the cluster. They distribute Hadoop MapReduce tasks more effectively to each heterogeneous node and determine the Hadoop processing mode (the parallel or sequential processing mode) depending on the computing resources for each node.

A native Hadoop framework, unlike map task assignment, does not provide specific policies for assigning the AM and reduce tasks to cluster nodes, causing significant Hadoop performance variations. This outcome significantly affects Hadoop performance, particularly on the performance-frugal SBC clusters. To resolve this problem, this paper also proposes new policies for effective AM and reduce task assignment across SBC cluster nodes according to the computational capabilities of each SBC node.

Consequently, by carefully considering the heterogeneity of SBC Hadoop cluster nodes, the redesigned Hadoop framework notably improves performance. To the best of our knowledge, the proposed Hadoop design is the first Hadoop YARN framework to explore various challenges on heterogeneous SBC Hadoop clusters for big data processing. Our comprehensive experiments with representative Hadoop benchmarks demonstrate that our redesigned framework achieves an average of 2.55× and 1.55× performance improvement under I/O-intensive Terasort workloads and CPU-intensive Wordcount workloads, respectively.

Our future work covers developing optimal JVM memory configurations for each container based on computing resources and workload characteristics.

## CRediT authorship contribution statement

**Sooyoung Lim:** Conceptualization, Data curation, Software, Visualization, Writing – original draft. **Dongchul Park:** Funding acquisition, Project administration, Supervision, Validation, Writing – review & editing.

## Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Dongchul Park reports financial support was provided by Korea Ministry of Science and ICT. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

No data was used for the research described in the article.

## References

[1] Y. Tang, H. Guo, T. Yuan, Q. Wu, X. Li, C. Wang, X. Gao, J. Wu, OEHadoop: Accelerate Hadoop applications by co-designing Hadoop with data center network, IEEE Access 6 (2018) 25849–25860, http://dx.doi.org/10.1109/ACCESS.2018.2830799.

[2] X. Ling, Y. Yuan, D. Wang, J. Liu, J. Yang, Joint scheduling of MapReduce jobs with servers: Performance bounds and experiments, J. Parallel Distrib. Comput. 90–91 (2016) 52–66, http://dx.doi.org/10.1016/j.jpdc.2016.02.002.

[3] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, D. Chen, G-Hadoop: MapReduce across distributed data centers for data-intensive computing, Future Gener. Comput. Syst. 29 (3) (2013) 739–750, http://dx.doi.org/10.1016/j.future.2012.09.001.

[4] K. Oh, M. Zhang, A. Chandra, J. Weissman, Network cost-aware geo-distributed data analytics system, IEEE Trans. Parallel Distrib. Syst. 33 (6) (2022) 1407–1420, http://dx.doi.org/10.1109/TPDS.2021.3108893.

[5] X. Lu, N.S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, D.K. Panda, High-performance design of Hadoop RPC with RDMA over InfiniBand, in: 2013 42nd International Conference on Parallel Processing, 2013, pp. 641–650, http://dx.doi.org/10.1109/ICPP.2013.78.

[6] H. Jiang, Y. Chen, Z. Qiao, T.-H. Weng, K.-C. Li, Scaling up MapReduce-based big data processing on multi-GPU systems, Cluster Comput. 18 (2015) 369–383, http://dx.doi.org/10.1007/s10586-014-0400-1.

[7] Y. Chen, Z. Qiao, S. Davis, H. Jiang, K.-C. Li, Pipelined multi-gpu mapreduce for big-data processing, in: Computer and Information Science, Springer, 2013, pp. 231–246, http://dx.doi.org/10.1007/978-3-319-00804-2_17.

[8] E. Lee, H. Oh, D. Park, Big data processing on single board computer clusters: Exploring challenges and possibilities, IEEE Access 9 (2021) 142551–142565, http://dx.doi.org/10.1109/ACCESS.2021.3120660.

[9] K. Neshatpour, M. Malik, M.A. Ghodrat, A. Sasan, H. Homayoun, Energy-efficient acceleration of big data analytics applications using FPGAs, in: 2015 IEEE International Conference on Big Data, Big Data, 2015, pp. 115–123, http://dx.doi.org/10.1109/BigData.2015.7363748.

[10] D. Park, J. Wang, Y.-S. Kee, In-storage computing for Hadoop MapReduce framework: Challenges and possibilities, IEEE Trans. Comput. (2016) 1–14, http://dx.doi.org/10.1109/TC.2016.2595566.

[11] S.J. Johnston, P.J. Basford, C.S. Perkins, H. Herry, F.P. Tso, D. Pezaros, R.D. Mullins, E. Yoneki, S.J. Cox, J. Singer, Commodity single board computer clusters and their applications, Future Gener. Comput. Syst. 89 (2018) 201–212, http://dx.doi.org/10.1016/j.future.2018.06.048.

[12] B. Qureshi, A. Koubâa, On energy efficiency and performance evaluation of single board computer based clusters: A hadoop case study, Electronics 8 (2) (2019) 182, http://dx.doi.org/10.3390/electronics8020182.

[13] BAE Systems, What are single-board computers? 2023, https://www.baesystems.com/en-us/definition/what-are-single-board-computers. (Last accessed 11 April 2023).

[14] P.J. Basford, S.J. Johnston, C.S. Perkins, T. Garnock-Jones, F.P. Tso, D. Pezaros, R.D. Mullins, E. Yoneki, J. Singer, S.J. Cox, Performance analysis of single board computer clusters, Future Gener. Comput. Syst. 102 (2020) 278–291, http://dx.doi.org/10.1016/j.future.2019.07.040.

[15] S.J. Cox, J.T. Cox, R.P. Boardman, S.J. Johnston, M. Scott, N.S. O'brien, Iridis-pi: a low-cost, compact demonstration cluster, Cluster Comput. 17 (2014) 349–358, http://dx.doi.org/10.1007/s10586-013-0282-7.

[16] P.J. Basford, G.M. Bragg, J.S. Hare, M.O. Jewell, K. Martinez, D.R. Newman, R. Pau, A. Smith, T. Ward, Erica the rhino: A case study in using raspberry pi single board computers for interactive art, Electronics 5 (3) (2016) 35, http://dx.doi.org/10.3390/electronics5030035.

[17] C. Pahl, S. Helmer, L. Miori, J. Sanin, B. Lee, A container-based edge cloud PaaS architecture based on Raspberry Pi clusters, in: 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops, FiCloudW, 2016, pp. 117–124, http://dx.doi.org/10.1109/W-FiCloud.2016.36.

[18] R. Zwetsloot, Raspberry Pi 4 specs and benchmarks, 2019, https://magpi.raspberrypi.com/articles/raspberry-pi-4-specs-benchmarks. (Accessed 23 July 2023).

[19] R. Gu, K. Huang, Z. Zhang, C. Yuan, Y. Huang, Push-based network-efficient hadoop YARN scheduling mechanism for in-memory computing, in: 2019 IEEE 25th International Conference on Parallel and Distributed Systems, ICPADS, 2019, pp. 133–140, http://dx.doi.org/10.1109/ICPADS47876.2019.00026.

[20] T. White, Hadoop: The Definitive Guide, fourth ed., ÓReilly Media Inc., Sebastopol, CA, USA, 2015.

[21] C. Kaewkasi, W. Srisuruk, A study of big data processing constraints on a low-power hadoop cluster, in: 2014 International Computer Science and Engineering Conference, ICSEC, 2014, pp. 267–272, http://dx.doi.org/10.1109/ICSEC.2014.6978206.

[22] A.J.A. Neto, J.A.C. Neto, E.D. Moreno, The development of a low-cost big data cluster using Apache Hadoop and Raspberry Pi. A complete guide, Comput. Electr. Eng. 104 (2022) 108403, http://dx.doi.org/10.1016/j.compeleceng.2022.108403.

[23] M. Zaharia, A. Konwinski, A.D. Joseph, R.H. Katz, I. Stoica, Improving MapReduce performance in heterogeneous environments, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI '08, USENIX Association, 2008, pp. 29–42, https://www.usenix.org/legacy/events/osdi08/tech/full_papers/zaharia/zaharia.pdf.

[24] Q. Chen, D. Zhang, M. Guo, Q. Deng, S. Guo, SAMR: A self-adaptive MapReduce scheduling algorithm in heterogeneous environment, in: 2010 10th IEEE International Conference on Computer and Information Technology, 2010, pp. 2736–2743, http://dx.doi.org/10.1109/CIT.2010.458.

[25] X. Sun, C. He, Y. Lu, ESAMR: An enhanced self-adaptive MapReduce scheduling algorithm, in: 2012 IEEE 18th International Conference on Parallel and Distributed Systems, 2012, pp. 148–155, http://dx.doi.org/10.1109/ICPADS.2012.30.

[26] K.L. Bawankule, R.K. Dewang, A.K. Singh, Historical data based approach to mitigate stragglers from the reduce phase of MapReduce in a heterogeneous Hadoop cluster, Cluster Comput. 25 (5) (2022) 3193–3211, http://dx.doi.org/10.1007/s10586-021-03530-x.
[27] N.S. Naik, A. Negi, T.B. B.R., R. Anitha, A data locality based scheduler to enhance MapReduce performance in heterogeneous environments, Future Gener. Comput. Syst. 90 (2019) 423–434, http://dx.doi.org/10.1016/j.future.2018.07.043.
[28] J. Dharanipragada, S. Padala, B. Kammili, V. Kumar, Tula: A disk latency aware balancing and block placement strategy for Hadoop, in: 2017 IEEE International Conference on Big Data, Big Data, 2017, pp. 2853–2858, http://dx.doi.org/10.1109/BigData.2017.8258253.
[29] M. Hammoud, M.F. Sakr, Locality-aware reduce task scheduling for MapReduce, in: 2011 IEEE Third International Conference on Cloud Computing Technology and Science, 2011, pp. 570–576, http://dx.doi.org/10.1109/CloudCom.2011.87.
[30] Q. Zhang, M.F. Zhani, Y. Yang, R. Boutaba, B. Wong, PRISM: fine-grained resource-aware scheduling for MapReduce, IEEE Trans. Cloud Comput. 3 (2) (2015) 182–194, http://dx.doi.org/10.1109/TCC.2014.2379096.
[31] K. Kc, V.W. Freeh, Dynamically controlling node-level parallelism in Hadoop, in: 2015 IEEE 8th International Conference on Cloud Computing, 2015, pp. 309–316, http://dx.doi.org/10.1109/CLOUD.2015.49.
[32] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, E. Ayguade, Resource-aware adaptive scheduling for MapReduce clusters, in: ACM/IFIP International Middleware Conference, 2011, pp. 187–207, http://dx.doi.org/10.1007/978-3-642-25821-3_10.
[33] B. Sharma, R. Prabhakar, S.-H. Lim, M.T. Kandemir, C.R. Das, MROrchestrator: A fine-grained resource orchestration framework for MapReduce clusters, in: 2012 IEEE Fifth International Conference on Cloud Computing, 2012, pp. 1–8, http://dx.doi.org/10.1109/CLOUD.2012.37.
[34] C.-H. Chen, J.-W. Lin, S.-Y. Kuo, MapReduce scheduling for deadline-constrained jobs in heterogeneous cloud computing systems, IEEE Trans. Cloud Comput. 6 (1) (2015) 127–140, http://dx.doi.org/10.1109/TCC.2015.2474403.
[35] A. Jabbari, F. Masoumiyan, S. Hu, M. Tang, Y.-C. Tian, A cost-efficient resource provisioning and scheduling approach for deadline-sensitive MapReduce computations in cloud environment, in: IEEE 14th International Conference on Cloud Computing, CLOUD, 2021, pp. 600–608, http://dx.doi.org/10.1109/CLOUD53861.2021.00078.
[36] J.-W. Lin, J. Arul, C.-Y. Lin, Joint deadline-constrained and influence-aware design for allocating MapReduce jobs in cloud computing systems, Cluster Comput. 22 (2019) 6963–6976, http://dx.doi.org/10.1007/s10586-018-1981-x.
[37] R. Li, Q. Yang, Y. Li, X. Gu, W. Xiao, K. Li, HeteroYARN: A heterogeneous FPGA-accelerated architecture based on YARN, IEEE Trans. Parallel Distrib. Syst. 31 (12) (2020) 2968–2980, http://dx.doi.org/10.1109/TPDS.2019.2905201.
[38] M. Lin, S. Chen, Flash-aware linux swap system for portable consumer electronics, IEEE Trans. Consum. Electron. 58 (2) (2012) 419–427, http://dx.doi.org/10.1109/TCE.2012.6227442.
[39] O. Kwon, K. Koh, Swap space management technique for portable consumer electronics with NAND flash memory, IEEE Trans. Consum. Electron. 56 (3) (2010) 1524–1531, http://dx.doi.org/10.1109/TCE.2010.5606292.
[40] J. Wang, M. Qiu, B. Guo, Z. Zong, Phase-reconfigurable shuffle optimization for hadoop MapReduce, IEEE Trans. Cloud Comput. 8 (2) (2020) 418–431, http://dx.doi.org/10.1109/TCC.2015.2459707.
[41] Y. Guo, J. Rao, D. Cheng, X. Zhou, iShuffle: Improving hadoop performance with shuffle-on-write, IEEE Trans. Parallel Distrib. Syst. 28 (6) (2017) 1649–1662, http://dx.doi.org/10.1109/TPDS.2016.2587645.
[42] D. Yang, X. Zhong, D. Yan, F. Dai, X. Yin, C. Lian, Z. Zhu, W. Jiang, G. Wu, NativeTask: A Hadoop compatible framework for high performance, in: 2013 IEEE International Conference on Big Data, 2013, pp. 94–101, http://dx.doi.org/10.1109/BigData.2013.6691703.
[43] H. Zhang, H. Huang, L. Wang, MRapid: An efficient short job optimizer on Hadoop, in: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2017, pp. 459–468, http://dx.doi.org/10.1109/IPDPS.2017.100.
[44] A. Adnan, Z. Tahir, M.A. Asis, Performance evaluation of single board computer for Hadoop distributed file system (HDFS), in: 2019 International Conference on Information and Communications Technology, ICOIACT, 2019, pp. 624–627, http://dx.doi.org/10.1109/ICOIACT46704.2019.8938434.

**Sooyoung Lim** received the bachelor degree in the Department of Software from Sookmyung Women's University, Seoul, South Korea, in 2022. She is currently pursuing the master's degree in computer science at Sookmyung Women's University. She is currently conducting research on efficient stack distance approximation algorithm design and computational capability-based optimal Hadoop JVM configurations. Her research interests include big data processing software platforms, cloud computing, heterogeneous computing memory sharing technologies and applications including Compute Express Link (CXL), storage systems including NVM-based SSDs.

**Dongchul Park** received the Ph.D. degree in computer science and engineering from the University of Minnesota–Twin Cities, Minneapolis, USA in 2012. From 2012 to 2017, he was a research engineer at Samsung, San Jose, CA and Intel, Hillsboro, OR, USA. From 2017 to 2022, he was an assistant professor in computer science at Hankuk University of Foreign Studies and Sookmyung Women's University, Seoul, Korea. Since 2023, he has been an associate professor in department of industrial security at Chung-Ang University, Seoul, Korea. His research interests focus on storage system design and applications, big data platforms, and Non-Volatile Memory.