Automatic Internal Parallelism Reconfiguration on Heterogeneous Low-Power Hadoop Clusters*

Sooyoung Lim, Dongchul Park*

Department of Industrial Security, Chung-Ang University, Seoul 06974, South Korea

Abstract

Apache Hadoop has been widely adopted for massive data processing and analysis for over a decade, using powerful, energy-consuming server clusters. Recently, energy efficiency and capital costs have become primary concerns of data centers; hence, low-power computers, such as single board computer (SBC) clusters, have been of interest as practical alternatives. These low-cost, low-power SBC clusters on the Hadoop platform introduce new problems due to the limited computational capability. The heterogeneity of SBC clusters is inevitable after adding newer nodes or replacing older or inactive nodes, creating more challenges because the native Hadoop framework does not carefully consider computational discrepancies in each node. This paper redesigns the Yet Another Resource Negotiator (YARN) architecture with Hadoop using intranode parallelism effectively. Unlike the original Hadoop YARN, the proposed design automatically reconfigures the number of concurrently processable tasks (i.e., concurrent containers) based on the actual computing resource information each node provides. Thus, this framework judiciously applies node-level parallelism in Hadoop according to workload characteristics. In addition, the redesigned YARN provides effective Hadoop task distribution policies, particularly for heterogeneous, low-power SBC clusters. The extensive experiments with Hadoop benchmarks demonstrate that the redesigned YARN framework performs better than the original framework by an average of 15% under I/O-intensive workloads and 6% under CPU-intensive workloads.

Keywords:
Big data, Hadoop, MapReduce, YARN, heterogeneous cluster, Raspberry Pi, SBC

1. Introduction

The Apache Hadoop platform has been extensively adopted across diverse industries and academic domains for over a decade as a central framework for storing and processing large datasets. Most companies have employed high-performance server clusters as the primary Hadoop infrastructure to best employ their high computational capability [1, 2, 3].

However, not all industries or academic fields require such powerful, general-purpose computing cluster environments due to their capital costs. Moreover, energy efficiency has recently become a significant concern in many companies [4, 5, 6, 7]. Thus, single board computers (SBCs), such as Raspberry Pi (RPi), have attracted substantial attention for constructing low-power, low-cost computing systems [8, 9]. Their energy efficiency and low cost make SBCs good option for embedded controllers to operate diverse complex devices, and they have evolved into more powerful minicomputers [10]. Thus, many researchers have explored the possibilities and opportunities for SBC clusters in big data processing and edge computing [9, 11, 12, 13, 14].

The Hadoop MapReduce framework on SBC clusters creates challenges due to the limited resources of each SBC node. With the suggested configurations of the native Hadoop platform, Hadoop does not exhibit optimal performance on resource-constrained SBC clusters because it does not effectively apply intranode parallelism. This node-level parallelism on SBC clusters must be carefully considered to achieve the best performance according to workload characteristics. Concurrently processing more tasks for each SBC node does not necessarily produce better performance. Lowering node-level parallelism can improve overall Hadoop MapReduce performance depending on the workload. However, the original Hadoop platform provides no convenient mechanism for effectively configuring node-level parallelism. Instead, users must manually configure systems by checking complicated parameters from Hadoop configuration files (e.g., mapred-site.xml, yarn-site.xml, etc.). Moreover, with a low memory capacity, SBCs frequently become inactive regardless of their cen-

*This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the Convergence security core talent training business support program(IITP-2025-RS-2023-00266605) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation).

*Dongchul Park is the corresponding author

Email addresses: stella161@cau.ac.kr (Sooyoung Lim), dongchul@cau.ac.kr (Dongchul Park)

tral processing unit (CPU) performance and eventually fail to complete MapReduce jobs. The native Hadoop framework does not provide an effective mechanism to detect such an inactive (i.e., overloaded) node accurately in the SBC cluster, and it continues assigning excessive tasks to them, causing job failure.

Heterogeneous SBC Hadoop clusters raise additional challenging problems due to the unequal computing capability among each node. The heterogeneity of cluster nodes over time is inevitable due to upgrading older nodes or replacing inactive ones [15, 16]. The original Hadoop platform does not carefully consider this heterogeneity problem of cluster nodes. Unlike map task distribution policies, it does not provide specific rules to distribute the reduce tasks and application masters (AMs) across nodes in the cluster. Thus, depending on their locations in the SBC Hadoop cluster, Hadoop MapReduce applications exhibit a significant performance variation for each identical operation, causing inconsistent (i.e., unstable) job execution times. Further, these applications frequently fail to complete the jobs because weak nodes cannot process the overloaded workloads that Hadoop assigns. Most existing research studies have aimed to address these problems under typical Hadoop cluster environments with sufficient computing power.

This paper presents the redesign of a Hadoop Yet Another Resource Negotiator (YARN) architecture to overcome the limitations of resource-frugal, heterogeneous SBC cluster environments. The architecture locally collects the computational capability of each SBC node and determines their appropriate node-level parallelism. Thus, we newly implement the NodeManager (NM) and ResourceManager (RM) components in YARN. This redesigned Hadoop framework investigates the computational capability of each node in advance and *automatically and immediately* applies the appropriate configurations (i.e., the number of concurrently executable tasks) to each SBC node by overriding preconfigurations. The native Hadoop requires a complicated manual preconfiguration process via external configuration files before Hadoop platforms start. This requirement implies that the native Hadoop framework cannot change the node-level parallelism of each node until the Hadoop systems stop, even though the configurations are not optimal for Hadoop frameworks and applications. Furthermore, the correct configurations heavily depend on user experience and knowledge.

In addition, Hadoop YARN scheduling policies are redesigned to resolve the severe performance variation problem of Hadoop MapReduce applications on heterogeneous SBC clusters. Whereas the mentioned node-level parallelism is primarily for map tasks, this YARN scheduler design is only applied to reduce tasks. The original Hadoop framework does not provide specific policies for reduce task assignments because powerful servers or computers have been adopted for typical Hadoop systems. The policies do not need to carefully consider reduce task locations across cluster nodes due to the abundant computing resources. However, their locations tremendously affect the overall Hadoop MapReduce application performance on resource-scanty, heterogeneous SBC clusters, noticeably varying the performance. Therefore, the proposed YARN scheduler assigns the reduce tasks to powerful SBC nodes based on the *actual* computing resource information that each node provides. Consequently, the proposed framework exhibits consistently high Hadoop MapReduce performance.

The main contributions of this paper are as follows.

Automatic Reconfiguring Mechanism: The Hadoop NM component is reimplemented to identify the actual computing resources of each node correctly in the SBC cluster. The Hadoop RM component is also redesigned to reconfigure the number of concurrently processable tasks automatically based on the computational capability each NM provides. Thus, the proposed design can effectively adjust node-level parallelism even after the Hadoop system starts by overriding the user preconfigurations.

Workload Effect Findings and Suggestions: High Hadoop intranode parallelism generally performs well. However, a MapReduce application with lower node-level parallelism performs better under heterogeneous SBC Hadoop clusters. In addition, we explore the correlation between the input data size and optimal node-level parallelism according to workloads. Our studies suggest that optimal configurations of the intranode parallelism should consider workload characteristics, particularly under heterogeneous SBC cluster environments.

The rest of this paper is organized as follows. Section 2 presents the related previous studies. Next, Section 3 details the design challenges and proposed solutions. Then, Section 4 describes the experimental results and analyses. Finally, Section 5 concludes this paper with the directions for future work.

2. Background and Related Work

This section explains Raspberry Pi (RPi) and details the Apache Hadoop YARN. In addition, it discusses the related work.

2.1. Raspberry Pi

The RPi is the most famous single board computer (SBC), and despite the advent of other SBCs, such as Gumstix, LattePanda, and BeagleBone [17], it quickly emerged as a frontrunner in the SBC market. Currently, RPi is the third best seller in general-purpose computers worldwide, following Apple and Windows computers [18]. Though it was initially intended to encourage students to learn basic computer science in schools by participating in electronics and programming [17, 19, 20], it has now been adopted across diverse industries, such as smart homes, robotics, manufacturing, medical, and even military applications [21].

Although the RPi is a cost-effective, energy-efficient computing platform, its low computational capability is a

Table 1: Specifications of Raspberry Pi (third and fourth generation) [9].

	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	1.4 W(idle), 3.7 W(full-load)	1.9 W(idle), 5.1 W(full-load)	2.7 W(idle), 6.4 W(full-load)
Release	February 2016	March 2018	June 2019
Price	$35	$35	$35(1 GB), $45(2 GB), $55(4 GB), $75(8 GB)

significant drawback. However, the fourth generation RPi (i.e., RPi 4 Model B) enabled processing large-scale data (i.e., more than hundreds of gigabytes) due to its performance improvement [9]. Table 1 presents the hardware specifications of the third generation RPi (RPi 3B and 3B+) and fourth generation RPi (RPi 4B). Compared with third generation RPi, the computing power of RPi 4B is significantly improved with 2.5× faster quad-core CPUs, up to 8 GB of RAM, UBS 3.0 ports, and a 1 Gb Ethernet network [22]. Therefore, Hadoop performance on the RPi 4B cluster achieved an average of 2× better performance than the RPi 3B cluster [9]. Furthermore, the unprecedented performance improvement of RPi 4B enabled its clusters to achieve performance comparable to a single desktop in the following two aspects: performance-per-dollar and performance-per-watt [9].

2.2. Apache Hadoop YARN

Apache Hadoop YARN (also known as MapReduce 2) is responsible for resource management and task scheduling in the Hadoop framework. This framework was introduced to address the following limitations of the original MapReduce framework: inefficient failure handling and the fault tolerance mechanism. To resolve these problems, YARN decouples cluster resource management and job scheduling from the data processing component of MapReduce.

The YARN framework comprises three key components: ResourceManager (RM), NodeManager (NM), and ApplicationMaster (AM). The RM is a single, global component that manages jobs and resources across the cluster. In a cluster, each worker node has a respective NM that monitors the node status and launches the MapReduce task on that node. The AM is created for each application and orchestrates the operation of each application in the cluster by working with the NM to execute and monitor tasks. In addition, YARN manages the launching of each AM and MapReduce task using a logical bundle unit called a container [23].

2.2.1. Containers

A container is a virtual execution environment for running YARN applications. The container is a fundamental resource allocation unit with Random Access Memory (in megabytes) and CPU cores to execute a single MapReduce task or the AM on the node. Diverse user-configurable Hadoop YARN properties exist for setting up containers (Table 2). Among them, the main control parameters are memory-relevant properties for calculating the container size. By default, Hadoop YARN adopts the *CapacityScheduler* using the DefaultResourceCalculator to calculate the resources needed to run an application. Because this approach solely considers memory when determining the number of containers, users have another option, the DominantResourceCalculator, which considers both memory and CPU for resource calculation.

Table 2: Hadoop configurations related to container settings [23]

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	-Xmx200m
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

The number of concurrent containers (CCs) indicates the number of simultaneous executable tasks in a node [24]. This value is directly calculated for each worker node using the following equations:

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

$$\begin{aligned} &yarn.scheduler.minimum{-}allocation{-}mb \\ &\leq mapreduce.map(or\ reduce).memory.mb \\ &\leq yarn.scheduler.maximum{-}allocation{-}mb \end{aligned} \quad (2)$$

Based on Equation 1, the number of CCs corresponds to the quotient obtained by dividing the total resources for each NM by the required resources for each MapReduce task. If the required resources for MapReduce tasks are below the minimum allocation unit, the CapacityScheduler automatically allocates its smallest unit for containers. However, it should not exceed the maximum allocation unit; otherwise, Hadoop triggers job failure (Equation 2).

The RM can assign up to eight CCs for each worker node by default. The total CC count across the cluster is called a task wave, representing the maximum number of concurrently executable containers in the cluster [25]. A wave factor is derived by dividing the total data chunk count by the task wave.

2.2.2. YARN Scheduling Mechanism

When a MapReduce job is submitted to the JobClient, it requests a new application creation from the RM. Be-

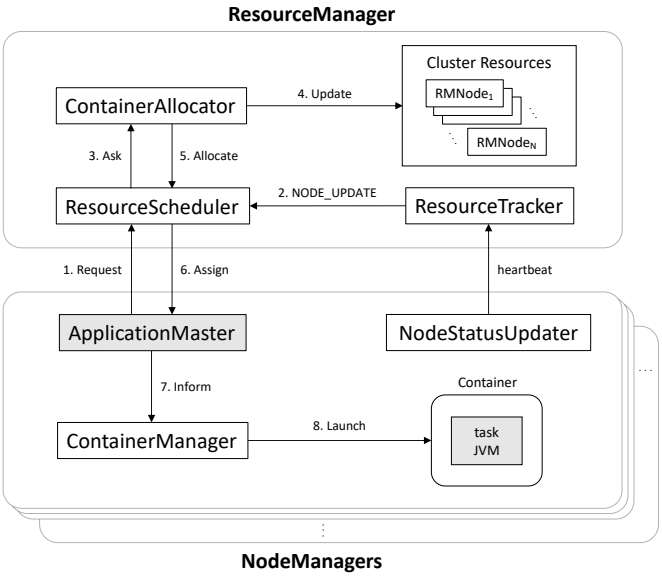


Figure 1: The overall YARN task scheduling process for launching each MapReduce task

cause the AM orchestrates the requests for scheduling each MapReduce task, the RM negotiates the creation of a container to initiate the AM before executing MapReduce tasks. The ApplicationsManager component in the RM begins negotiations with the ResourceScheduler to allocate an AM container. The AM container is the first container to be assigned. Thus, it is scheduled to *any* node that sends the earliest heartbeat to the ResourceTracker. Following this bootstrap phase, the AMLauncher component informs the corresponding NM to launch the AM container for each submitted application.

Figure 1 depicts the overall MapReduce task scheduling processes of Hadoop YARN. This process is identical to an AM execution mechanism except for the entity responsible for the container request (i.e., the RM and AM). The AM requests containers from the RM via periodic heartbeats. These heartbeats contain a list of each container's locality information and the container priority. Upon receiving these heartbeats, the RM awaits a heartbeat from an NM according to the locality preference. Once an NM is available for resource allocation, it informs the RM of its status via the NodeStatusUpdater component. Then, the ResourceTracker triggers a NODE_UPDATE event to the ResourceScheduler, asking the ContainerAllocator to allocate resources based on the locality preference. Moreover, the AM periodically sends heartbeats to the RM to request resources. Once the AM receives containers, each NM launches the assigned containers via the ContainerManager component.

2.3. Related Work

2.3.1. Hadoop on SBC Clusters

Several research studies on Apache Hadoop performance evaluation have been conducted on low-cost, low-performance SBC clusters.

Bourhnane *et al.* [11] built a cluster of five RPi 3B nodes and a single commodity computer to measure MapReduce performance (i.e., Terasort only) and power consumption. They discussed each performance regarding energy-efficiency and cost-effectiveness, concluding that the single personal computer outperformed the RPi cluster, and a cluster of 500 RPi nodes would provide comparable performance to the single personal computer.

Cubieboard is another single board computer. Kaewkasi and Srisuruk [26] built a Hadoop cluster of 22 cubieboards, where each node has an ARM Cortex A8 processor. They evaluated the performance of Apache Spark and claimed the cluster could successfully process 34 GB of a Wikipedia article file in an acceptable time. Moreover, it consumed low power (0.061 to 0.322 kWh) under all benchmarks. They concluded that low input/output (I/O) and CPU processing capability were the main cluster performance bottlenecks.

Lee *et al.* [9] conducted comprehensive Hadoop performance studies on the latest generation RPi cluster. They set up a Hadoop cluster of five most powerful RPi 4B nodes (with 4 GB of RAM each) and evaluated various Apache Hadoop and Spark benchmarks. In addition, using the USB 3.0 port, they explored the performance effects of various storage media on the Hadoop performance. They found that the latest generation RPi 4B cluster can provide sufficient performance to process even 500 GB of actual big data.

Recently, Neto *et al.* [27] provided a complete guide to developing, testing, and monitoring of SBC Hadoop clusters. Like Lee *et al.* [9], they also built a Hadoop cluster of nine RPi 4B nodes and performed Terasort and TestDFSIO benchmarks with various data sizes (250 MB up to 1 GB). Further, Neto *et al.* adopted one RPi 3B+ node as a monitoring server, enabling collecting real-time information for cluster monitoring and visualization.

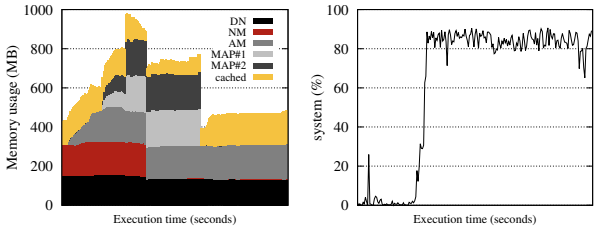
2.3.2. Hadoop Performance Optimization

The heterogeneity of cluster nodes is unavoidable over time when upgraded or replaced. Different computational capabilities of Hadoop cluster nodes create various problems, such as job delays and inefficient resource management. Thus, many research studies have been conducted to address the heterogeneity problem.

Paik *et al.* [28] and Xie *et al.* [29] focused on optimizing data distribution in heterogeneous environments by considering the capacities and performance levels of each node. They aimed to improve the data processing speed by strategically distributing tasks among nodes based on their capabilities.

Bawankule *et al.* [30] and Naik *et al.* [31] studied a computational skewness problem from the reduce phase. Both studies resolved it by scheduling each reduce task on a faster computing node in heterogeneous environments.

Htay and Phyu [32] analyzed the performance of MapReduce by varying the number of CCs per machine and sug-



(a) Memory usage (b) CPU utilization

Figure 2: Resource use over time on a single RPi 3B worker node processing 256 MB of data with two map tasks.

gested the optimal number of concurrent map tasks per node under Terasort and Wordcount. Janardhanan and Samuel [33] proposed the appropriate split sizes and estimated the job execution time for each level of parallelism by adopting a regression model.

Most existing research studies have addressed the heterogeneity problem on *general* computer or server-based heterogeneous Hadoop clusters. However, heterogeneity on *resource-frugal* SBC Hadoop clusters creates noticeably different challenging problems. This paper explores the node-level parallelism of YARN in the heterogeneous SBC Hadoop cluster.

3. Main Design

This section describes the research motivations and the proposed Hadoop YARN design to employ Hadoop's node-level parallelism effectively.

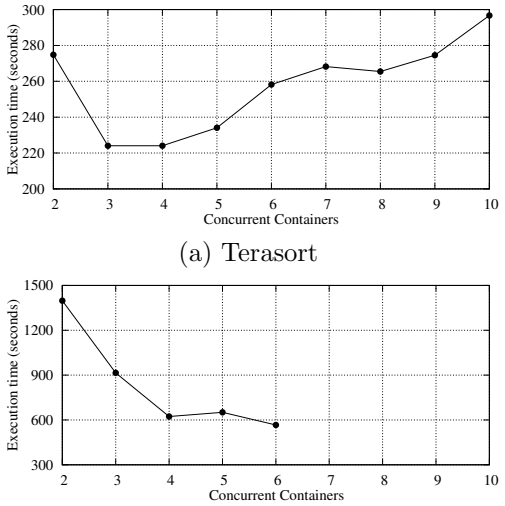
3.1. Motivations

3.1.1. Effect of Computing Resources

Each Hadoop map (or reduce) task runs on its task Java virtual machine (JVM), requiring a certain amount of memory. However, some SBCs cannot meet MapReduce tasks' resource requirements due to their scanty resources. Thus, Hadoop job failure is caused by an *out-of-memory* (OOM) problem.

Figure 2 illustrates this critical OOM problem during the Hadoop MapReduce job execution on an SBC worker node with low computational capability (e.g., a third generation RPi). We analyzed the system resource information during an OOM occurrence by Running native Hadoop framework on an RPi 3B node with two map tasks and 256 MB of data. Figure 2-(a) depicts a sudden drop in the total memory usage due to the NM loss (red) because the NM is interrupted when the number of map tasks exceeds the computational capability of the node. Figure 2-(b) displays the system-level CPU utilization of the corresponding RPi 3B node. The CPU utilization suddenly rises to about 90% when the OOM problem occurs, resulting from context switching overhead in each JVM.

Based on our comprehensive studies, physical memory capacity has a more critical influence on MapReduce job execution than CPU computational capability.



(a) Terasort

(b) Wordcount

Figure 3: Hadoop execution time by node-level parallelism (i.e., CC count) configuration. Each native Hadoop benchmark is executed on a single RPi 4B (4 GB RAM) worker node with a 1 GB dataset. Wordcount is not executable with more than six CCs on the single RPi 4B due to memory limitations.

3.1.2. Effect of Parallelism

For more effective processing, each Hadoop worker node concurrently executes multiple MapReduce tasks by selecting appropriate node-level parallelism according to the computational capability of each node. To optimize Hadoop MapReduce performance, Hadoop provides various configuration parameter sets that users can reconfigure based on their insight and experience.

Figure 3 presents the total execution time of Terasort and Wordcount on a single RPi 4B node according to the number of CCs. Interestingly, the higher level of parallelism (i.e., the more concurrently executable task counts) does not necessarily perform better on an SBC Hadoop cluster node (Figure 3-(a)). In addition, both applications have different optimal CC counts. Moreover, selecting an irrelevant CC count (Figure 3-(b)) or an SBC node with minimal computational capability (Section 3.1.1) causes the mentioned OOM error.

Based on these observations, the optimal Hadoop CC count must be carefully selected, rather than using a single CC count for all applications like native Hadoop configurations, according to workload characteristics (e.g., I/O-intensive, CPU-intensive, or mixed). The original Hadoop does not provide this intelligent feature.

3.1.3. Same Job, Different Performance

Hadoop map task processing is closely associated with data locality across the cluster. Unlike this map task, reduce tasks and the AM can be executed at any node when an identical Hadoop job is submitted to the Hadoop cluster because the native Hadoop has no specific policies for assigning reduce tasks and the AM in the cluster. This

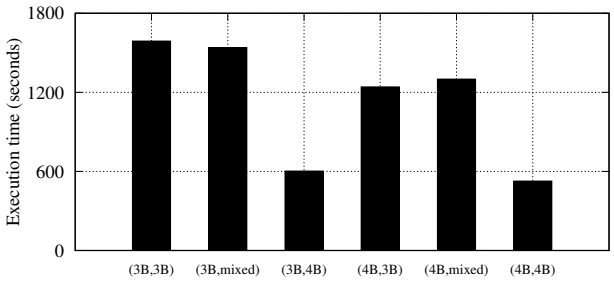


Figure 4: Hadoop Terasort benchmark performance (4 GB dataset) by distributions of both AM and two reduce tasks to RPi 3B or RPi 4B nodes in the heterogeneous SBC cluster. Mixed: two reduce tasks are individually assigned to RPi 3B and 4B.

critically affects Hadoop performance, particularly on heterogeneous SBC clusters.

Figure 4 illustrates a Hadoop performance according to the locations of the reduce tasks and AM on the heterogeneous SBC Hadoop cluster. This study demonstrates that the overall Hadoop performance is significantly inconsistent (with up to a 2.5× performance gap between the best and worst performance) even though the same job is repeatedly executed on the SBC cluster. This result implies that intelligent Hadoop reduce (or AM) task scheduling policies must be implemented to produce the best performance consistently.

3.2. Proposed Design: Overall Architecture

Figure 5 presents the redesigned Hadoop YARN architecture and describes the overall processes of updating *Cluster Resources*, the aggregation of each ResourceManagerNode (RMNode) component. The RM manages the information for each NM with the RMNode; thus, each RMNode is created in RM when each NM is registered on the RM. Each RMNode provides information on the available resources (i.e., memory, CPU, etc.) to the ResourceScheduler and keeps track of all containers running on its node. To allocate MapReduce tasks to each node effectively based on the available computing resources, every RMNode must handle its own *nodeResources*, representing the actual computational capability of each node.

During each NM initialization, the *nodeResources* collector component collects the computing resource information of a local SBC node, such as the physical memory capacity and number of CPU cores. Once the NM completes the setup process, it sends its *nodeResources* and other local information to the RM for registration. Right after the RM accepts the registration request from the NM, it generates the corresponding RMNode using the *nodeResources* information that the NM provides. Then, the RM verifies actual computing capability of each node via the *nodeResources* identifier. To determine MapReduce processing capability *correctly* for each node, the RM recalculates the number of concurrently executable containers (i.e., node-level parallelism) based on *nodeResources* and

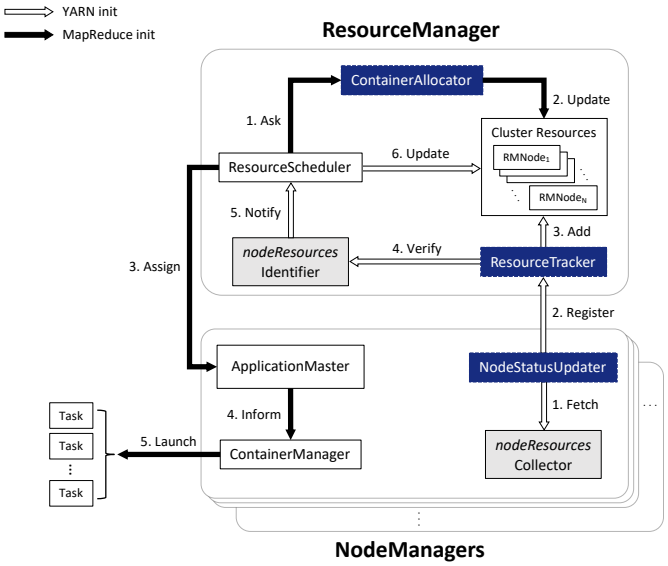


Figure 5: Redesigned Hadoop YARN architecture. Added (gray) or redesigned (blue) components are marked with solid squares. Please note the ApplicationMaster is created per application, not per node.

overrides the existing Hadoop configuration values that users initially configured before the Hadoop system starts. Finally, the RM notifies the ResourceScheduler of the recalculated CC information and updates the total resource information of the corresponding NM in the RMNode.

After initiating this YARN service, the RM must launch the AM before processing the MapReduce tasks because MapReduce task scheduling is closely associated with the AM component. The AM container is the first container allocated; thus, it is assigned to one of the powerful SBC nodes that sent the earliest heartbeat event to the RM. The powerful SBC nodes are identified by *nodeResources* in its RMNode. Once the AM is launched, Hadoop YARN performs MapReduce task scheduling processes. The ResourceTracker dispatches a STATUS_UPDATE event to the corresponding RMNode component, followed by a NODE_UPDA event to the ResourceScheduler. The ResourceScheduler requests the ContainerAllocator to allocate resources to each node and updates the corresponding RMNode accordingly. Containers are assigned to map tasks based on the default locality preference to execute map tasks. Similar to the proposed AM container allocation, the redesigned ContainerAllocator intelligently assigns each reduce container to the powerful (i.e., not simply any) SBC nodes that do not execute a reduce task. The SchedulerNode component of each node manages the number of reduce tasks running locally.

3.3. Design Challenges and Implementation

This section presents several design challenges and their respective implementations.

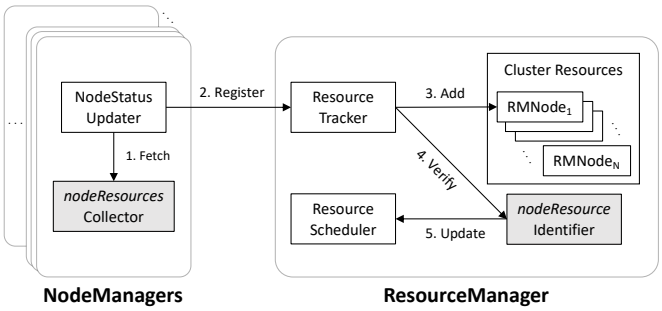


Figure 6: The *nodeResources* collector and identifier in Hadoop YARN. In each worker node, NodeManager fetches the correct computing resource information from the *nodeResources* Collector and sends it to the ResourceManager for registration.

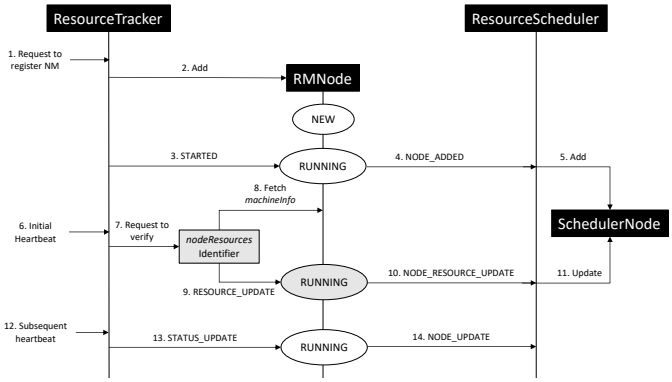


Figure 7: New process for checking the hardware specifications for each node.

3.3.1. Computational Capability Identification

The computing resources, such as physical memory capacity and CPU core counts, critically affect the concurrent MapReduce task execution. However, the native Hadoop framework does not provide an effective mechanism to consider the actual computational capability of each node. Figure 6 illustrates the proposed mechanism to collect the actual computing resource information for each SBC node and register it in YARN. The redesigned Hadoop YARN calculates an appropriate number of CCs based on this actual resource information that each *nodeResources* collector provides. When a Hadoop system starts, this proposed mechanism enables Hadoop YARN to select a more reasonable parallelism level on each resource-scanty SBC node by overriding the initial Hadoop configurations that users irrelevantly configured or the original Hadoop provided by default.

To redesign this YARN initialization process, we first implemented a Linux child process of NM on each worker node, the *nodeResources* collector, collecting the computing resource information for each node. The hardware specifications for each computing node are not typically changed when Hadoop systems are running; thus, this process is executed only once as soon as the NM starts to setup. Thus, each NM fetches actual computing resource information from the respective *nodeResources* collector component and sends it to the RM. We implemented a novel transmission standard, including *nodeResources*, using a protocol buffer application programming interface (API). This modified inter-process communication (IPC) protocol enables each NM to transmit *nodeResources* to the RM.

When the information from each NM is registered on the RM, *nodeResources* must also be included. That is, the RM manages the information for each NM in the RMNode component; hence, each RMNode is created in the RM when each NM is registered on the RM. Thus, we added *nodeResources* information for each NM in the corresponding RMNode properties.

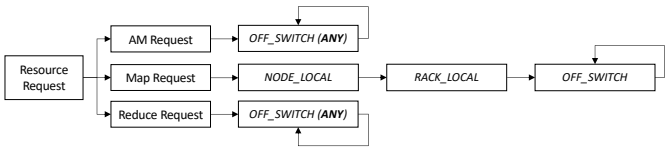
Based on the information each *nodeResources* collector provides, the RM verifies each node parallelism level via *nodeResources* identifier, considering the physical memory capacity and number of CPU cores. The RM first considers the physical memory capacity for detecting resource-frugal SBCs because our comprehensive study found that the memory space critically affects determining an ideal parallelism level. Then, the RM considers the CPU specifications (i.e., CPU core counts). After recalculating the relevant CC count for each node, the parallelism level for each node is updated and the information is sent to the ResourceScheduler to update the YARN resources according to the updated CC counts.

3.3.2. Updating Node-level Parallelism

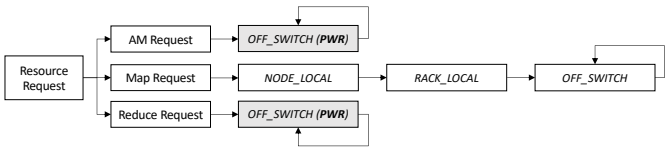
Figure 7 presents the following process after the NM registration in more detail, primarily the update process of each YARN resource based on *nodeResources*.

After the ResourceTracker accepts a request to register a new NM, it creates an RMNode component for managing the corresponding NM information, including *nodeResources*. Once the RMNode is added to the cluster node list, the ResourceTracker registers its information and dispatches the *STARTED* event to the RMNode to signify its state transition from NEW to RUNNING. Then, RMNode sends the *NODE_ADDED* event to the ResourceScheduler to create a SchedulerNode component containing the information for YARN scheduling resources, such as the total resources and number of allocated containers. From then on, the NM can periodically transmit its information to the RM via a heartbeat communication.

The physical computing environment must be considered by adjusting the degree of parallelism based on the *nodeResources* before submitting the MapReduce applications to prevent a Hadoop performance drop or job failure. The ResourceTracker initiates a verification request to check the parallelism level of each node by cross-referencing its actual hardware specifications in *nodeResources* via the *nodeResources* identifier in the RM. Upon receiving the initial heartbeat from the NM, the *nodeResources* identifier calculates the appropriate number of CCs based on its *nodeResources* fetched from its RMNode using the Equation 1. Then, it sends the *RESOURCE_UPDATE* event to



(a) Original ContainerAllocator



(b) Redesigned ContainerAllocator

Figure 8: Redesigned ContainerAllocator component in the ResourceManager. The proposed policy assigns the reduce tasks (or AM) to *powerful* SBC nodes, not just *any* SBC nodes in the heterogeneous SBC cluster.

the RMNode, prompting to transmit *NODE_RESOURCE_UPDATE* event to the ResourceScheduler. This process updates the node-level parallelism of Hadoop if the newly proposed CC count differs from the original one calculated from the existing Hadoop configurations.

Once this CC count update completes, the RM resumes its regular heartbeat process at a predefined interval (i.e., 1 second by default) until the NM terminates.

3.3.3. Effective Hadoop Task Distribution

Hadoop classifies each task into three levels of data locality. To optimize data processing, Hadoop prioritizes tasks in the following order: highest priority (NODE_LOCAL), intermediate priority (RACK_LOCAL), and lowest priority (OFF_SWITCH). Each ResourceRequest specifies these locality preferences. If allocation fails even at the lowest priority (OFF_SWITCH), Hadoop retries allocation within the same priority level until a container becomes processable. Figure 8-(a) exhibits the priorities of the original Hadoop tasks. Unlike map tasks, the AM and reduce tasks have the lowest priority, indicating that the original Hadoop distributes them to *any* (i.e., strong or weak) node in the cluster. The significant Hadoop performance variations (Figure 4) on heterogeneous SBC Hadoop clusters result from this Hadoop policy.

Moreover, although both the AM and reduce tasks are distributed to powerful SBC nodes, they may sometimes be assigned to the same node (worst case) or distributed to a separate node (best case) due to the transmission delay of a NODE_UPDATE heartbeat to the RM. The original Hadoop exhibits inconsistent performance according to their locations; hence, it cannot guarantee the best performance for each job execution.

For consistent and optimal Hadoop performance, a new data locality policy is proposed. Figure 8-(b) displays the modified *OFF_SWITCH* policy in Hadoop YARN's *ContainerAllocator* component. The proposed policy distributes both tasks to powerful (PWR) SBC nodes, not just any SBC node in the heterogeneous SBC cluster. Specifically, if a ResourceRequest corresponds to either the reduce task request or AM request, the redesigned ContainerAllocator assigns them to the SBC nodes with high computational capability.

In addition, the requests should be evenly allocated to powerful SBC nodes for the best performance. Thus, a new operation in the *SchedulerNode* component is implemented to manage the assigned reduce task count on its node. When a map phase reaches a predefined overall progress (e.g., 5% by default), the reduce containers in the SCHEDULED state are assigned to nodes without active reduce tasks. Once its state is changed from SCHEDULED to RUNNING, each SchedulerNode increments the assigned reduce task count by one.

This new operation guarantees optimal distribution of each reduce task to powerful SBC nodes (not overlapped). Thus, the proposed design effectively executes the reduce tasks for the best cluster-level parallelism on the heterogeneous SBC Hadoop cluster.

3.4. Baseline Design

Section 3.1.1 reveals that the SBC cluster nodes with a small RAM (e.g., 1 GB) cannot run original Hadoop MapReduce applications with a large data set. That is, the heterogeneous RPi cluster cannot complete well-known Hadoop benchmarks on memory-frugal SBC nodes because they consistently cause a node failure error due to the insufficient memory of the weak SBC nodes (e.g., RPi 3B or 3B+ with 1 GB of RAM). For an objective evaluation of the redesigned framework, we require a different (original) Hadoop framework that ensures successful job processing with massive data. Therefore, we *minimally* modified the original Hadoop framework for the baseline design. First, we fixed the number of CCs of the third generation (not the fourth or later) RPi to one; otherwise, they (RPi 3B or 3B+) cause a job failure error.

In addition, the AM and all reduce tasks can be distributed to any SBC node in the cluster because the original Hadoop framework has no specific rules for their distribution. Consequently, various combinations of the locations are observed due to their distributions across individual cluster nodes. As described in Section 3.3.3, this results in noticeable performance differences depending on the locations across heterogeneous SBC clusters, causing a critical problem for performance evaluation because when an identical Hadoop benchmark with an identical data set is run, it produces different performance results.

Therefore, we also applied the redesigned *ContainerAllocator* component to the baseline Hadoop framework for consistent performance. Please note the ContainerAllocator component consistently enables the baseline Hadoop to exhibit best (not worst) performance (out of all possible performance cases). Thus, this baseline Hadoop consistently guarantees better performance than the original (i.e., never touched) Hadoop. Thus, we *never manipulated the experiments to our advantage.*

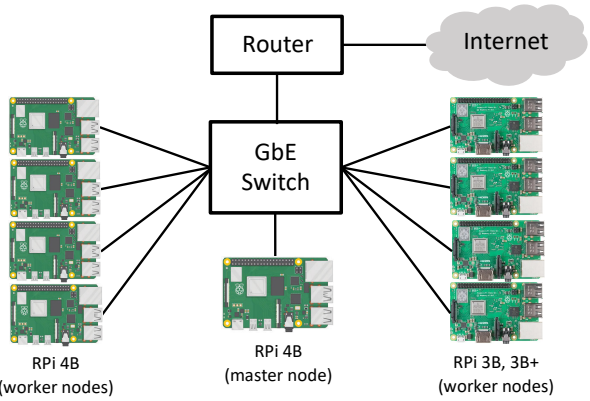


Figure 9: Our heterogeneous SBC Hadoop cluster architecture.

Table 3: Experimental setup configurations

	Master node	Worker nodes		
Node	RPi 4B (4 GB)	RPi 4B (4 GB)	RPi 3B+	RPi 3B
Number of nodes	1	4	1	3
OS	Ubuntu Desktop 22.04.3 LTS (ARM 64-bit)			
Storage media	MicroSD 256GB (Sandisk Extreme Pro)			

Table 4: Hadoop framework configurations

Framework	Apache Hadoop 3.2.3
ResourceScheduler	CapacityScheduler
HDFS chunk size	128MB
HDFS replication factor	3
Container heap size	820MB

We did not change any parts of the original Hadoop framework except for these two factors. The native Hadoop source codes were slightly modified for successful job processing and a fair evaluation with our proposed Hadoop framework on the heterogeneous SBC Hadoop cluster.

4. Performance Evaluation

This section presents the comprehensive experimental results and analyses.

4.1. Experimental Setup

4.1.1. Cluster Configurations

As in Figure 9, the heterogeneous SBC Hadoop cluster of nine nodes was built for our experiments: eight worker nodes (4×RPi 4B, 3×RPi 3B, and 1×RPi 3B+) and one master node (RPi 4B). Table 1 presents more detailed specifications of each RPi node.

The Hadoop cluster configurations are listed in Table 3. Due to the widely known stability and compatibility problems of the original Raspberry Pi Operating System (OS) [9], the Ubuntu 22.04 desktop OS (64-bit) was employed. Considering resource-scanty SBC nodes, we excluded the GNOME Graphic User Interface (GUI) from Ubuntu. Moreover, cooling fans and heat sinks were applied to all SBC nodes (particularly RPi 4B) to resolve their CPU thermal throttling problem [9].

Table 4 lists the Hadoop configurations. We adopted 820 MB of the upper limit of the JVM heap memory size (i.e., mapreduce.map(or reduce).java.opts), corresponding to about 80% of memory-relevant container settings because the default value consistently causes job failure (i.e., out-of-heap-memory error). Except for this, all default values of the Hadoop YARN configuration properties in Table 4 were employed for an objective evaluation.

4.1.2. Hadoop MapReduce Applications

The following three Hadoop benchmarks were employed for a comprehensive evaluation: Terasort (I/O-intensive workload), Grep and Wordcount(CPU-intensive workloads). These are very well-known Hadoop MapReduce applications.

Terasort randomly sorts generated data from the Teragen application. After reading the input data, the benchmark sorts them (the map phase) and writes the final output results (the reduce phase) with the Hadoop Distributed File System (HDFS). Unlike other typical Hadoop MapReduce applications, Terasort does not generate the final data with a reduced size because it just rearranges the input data sequence in descending or ascending order.

Wordcount counts the number of distinct word occurrences from input data. A map function splits the input data into each word with a key-value (i.e., word-count) pair format and generates intermediate (i.e., temporary) data. Finally, the reduce function aggregates the intermediate data of each map function to the final word count data in a sorted manner.

Grep searches user-defined patterns from input files and counts them. It performs both a grep-search job and a grep-sort job. The grep-search job executes the same operation as in Wordcount. After the grep-search completes, the grep-sort job creates a mapper per output file (map phase) and makes a final output file with a single reducer.

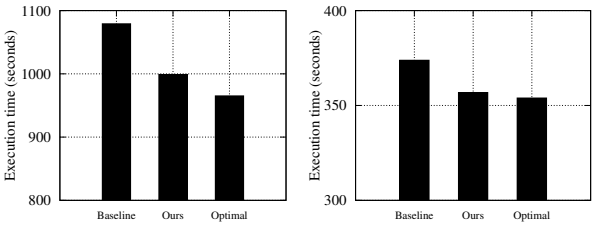
4.1.3. Performance Evaluation Metrics

The total elapsed time of the three aforementioned MapReduce applications was primarily measured. This metric represents the actual Hadoop application performance and was subdivided into the initial setup, map, reduce, and cleanup phases.

For a deeper analysis, the various system resources, such as CPU utilization (usr, sys, iowait), memory consumption, and I/O traffic, were comprehensively measured. However, other system resources (e.g., network traffic, power consumption, I/O requests and CPU temperature) were also evaluated. No difference was found between our redesigned and the original Hadoop YARN because they are not directly affected by the proposed design. Thus, this section omits them. Identical data sets and Hadoop configurations were applied for a fair evaluation. We executed

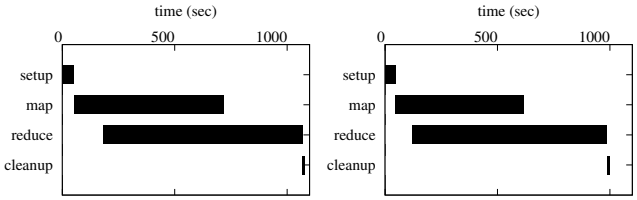
Table 5: Total execution times (seconds). Here, a reduce task count is 4.

Benchmark	Input data	# of Concurrent containers on RPi 4Bs						
		2	3	4	5	6	7	8
Terasort	2GB	172.3	157.4	157.8	155.9	157.9	168.3	171.7
	4GB	318.0	262.1	272.5	254.9	276.9	297.6	360.1
	8GB	593.3	527.5	495.6	482.4	515.7	534.7	555.9
	16GB	1137.5	989.1	999.0	965.1	993.0	1008.6	1079.1
	32GB	2320.4	2232.8	2153.2	2265.4	2237.7	2422.3	2441.8
	64GB	4217.9	3673.7	3679.3	3788.0	3932.0	4008.6	-
Grep	2GB	150.3	143.5	145.9	153.4	149.3	180.2	180.6
	4GB	226.0	189.0	177.2	188.1	179.4	175.3	191.5
	8GB	309.8	268.2	232.6	240.1	239.7	229.0	240.1
	16GB	534.5	398.9	356.6	359.4	355.2	353.9	373.8
	32GB	951.1	681.4	604.0	598.0	594.1	594.1	595.9
	64GB	1792.3	1249.1	1115.0	1083.7	1080.5	1082.5	1083.1
Wordcount	2GB	492.2	485.5	488.7	483.8	475.4	491.6	491.3
	4GB	1101.2	800.3	605.5	562.4	591.6	×	×
	8GB	2121.3	1341.4	1115.1	998.9	975.8	×	×
	16GB	4272.9	2611.4	2112.3	1763.5	1828.9	×	×
	32GB	8430.3	5124.3	3967.0	3541.5	3503.7	×	×
	64GB	16642.9	10249.5	7740.1	6913.8	6855.4	×	×



(a) Terasort (b) Grep

Figure 10: Total execution time for each Hadoop framework. Data size: 16 GB, reduce task count: 4.



(a) Baseline (b) Ours

Figure 11: Execution time breakdown for MapReduce tasks (16 GB and 4 reduce tasks with Terasort). Here, reduce phase includes shuffle phase.

sync, *drop cache*, and *trim* system commands for every iteration.

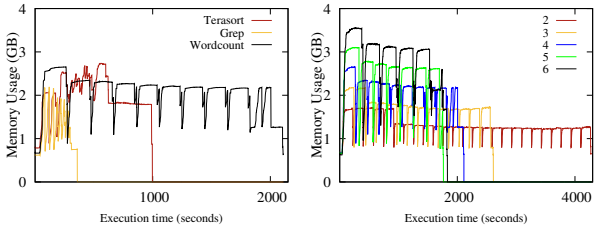
4.2. Experimental Results and Analyses

4.2.1. Overall Performance: Effect of Parallelism

Table 5 lists the overall performance (i.e., total execution time) of the three MapReduce benchmarks. The CC count on each RPi 4B node was varied from 2 to 8 to explore its influence (i.e., node-level parallelism) on the heterogeneous RPi Hadoop cluster. One of the RPi 4B nodes must execute one ApplicationMaster (AM); hence, the CC count starts with 2, not 1. Higher node-level parallelism does not necessarily produce better performance. For instance, Terasort with a CC count of 5 achieves an average of 1.12× better performance than that with a CC count of 8 (i.e., the original Hadoop configuration). The optimal node-level parallelism in the heterogeneous SBC Hadoop cluster tends to depend on workload characteristics. Section 4.2.2 and 4.2.3 discuss this in more detail.

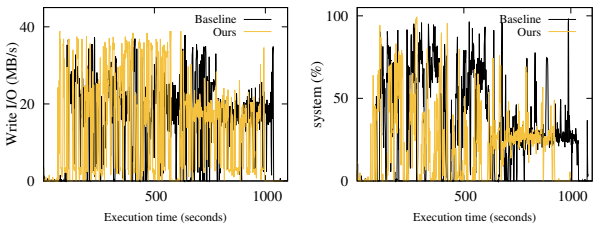
Figure 10 compares the overall performance of our proposed Hadoop framework to that of the baseline. Optimal performance is also presented for a more objective comparison. As in the figure, the proposed design achieves better performance than the baseline by an average of 15% and 6% under Terasort and Grep respectively. This improvement originates from a more effective node-level parallelism on RPi 4Bs, which highly affects the map phase.

Figure 11 breaks down the total execution time into four Hadoop execution phases for a deeper analysis. Each framework employs 16 GB of data and four reduce tasks under Terasort (Figure 10-(a)). Unlike other phases, the execution time of the map phase displays significant differences between the baseline and our proposed Hadoop framework. This performance gap primarily results from the effectiveness of node-level parallelism between the two designs. The map phase is closely associated with concurrently processing distributed input data across cluster nodes. For more effective map task processing on the heterogeneous SBC Hadoop cluster, the redesigned Hadoop automatically adjusts node-level parallelism based on the actual computing performance of each SBC node. Consequently, the proposed design performs better than the baseline design by an average of 15% and 6% under Terasort and Grep respectively. We equally applied the reduce task assignment policy to the baseline (to prevent job failure in the native Hadoop framework); thus, neither framework displays a noticeable performance difference. Similarly, each Hadoop framework has identical cleanup time because we did not change the cleanup algorithm.



(a) All workloads (b) Wordcount

Figure 12: Memory consumption by (a) workload on RPi 4B for the redesigned Hadoop framework and (b) concurrent container count for Wordcount on map task JVMs. Data size: 16 GB, reduce task count: 4.



(a) Write I/O traffic (b) CPU utilization (sys)

Figure 13: Resource utilization on RPi 4B nodes. Data size: 16 GB, reduce task count: 4.

Table 6: Total execution time (seconds) of I/O-intensive Wordcount with disabled combiner.

Benchmark	Input data	Concurrent containers on RPi 4Bs				
		2	3	4	5	6
Wordcount	2GB	2374.5	2147.7	2162.2	2109.0	2163.1
	4GB	4725.4	4679.2	4646.0	4819.9	4787.5
	8GB	9608.4	9221.7	9025.7	9398.1	9761.9
	16GB	21160.7	20030.6	20391.3	20887.7	×
	32GB	49875.6	46176.8	47326.7	49268.4	×
	64GB	×	×	×	×	×

Compared to the optimal performance, our proposed design has an average of 1.02× lower performance than the optimal performance under both workloads. On the other hand, the original Hadoop (i.e., baseline) exhibits an average performance of 1.18× and 1.08× lower than the optimal one, respectively. The proposed design performs close to the optimal performance.

More importantly, our design guarantees successful MapReduce job processing without the OOM (i.e., Out of Memory) problem even on memory-frugal SBC nodes. Figure 12-(a) displays the memory consumption for each workload. During the first task wave, Wordcount uses about 2.5 GB of RAM and the others (i.e., Terasort and Grep) consume 2 GB of RAM, implying that increasing the number of concurrent tasks is more impactful under Wordcount than Terasort or Grep. This result is the primary cause that Wordcount with more than 6 CCs cannot be executed on RPi 4B nodes with 4 GB of RAM (Table 5). Figure 12-(b) demonstrates the experimental results, depicting the memory that the Wordcount map task JVMs consume for each CC count. In the figure, Wordcount with a CC count of 6 on the RPi 4B node (4 GB of RAM) consumes more than 3.5 GB of RAM, meaning that Wordcount configured with 7 or more CCs cannot complete due to the OOM error.

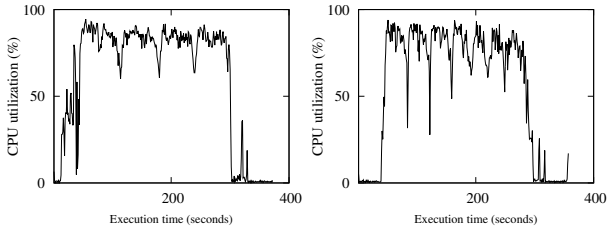
4.2.2. Effect of Workloads: I/O-intensive

This section explores the implications of workloads. Terasort is a representative I/O-intensive application because its sorting process involves frequent data transfers between RAM and external storage. As in Figure 13-(a), a significant amount of write I/O traffic is observed. The map phase constantly reaches the maximum write performance of the storage media card [9].

These I/O operations cause system-level interrupts and context switches, affecting CPU utilization. However, our proposed framework effectively adjusts the concurrency level of each worker nodes according to their computing capability, avoiding CPU contention. On the other hand, the baseline adopts excessive parallelism so that each RPi 4B node cannot effectively process MapReduce tasks because the original Hadoop configuration does not consider the actual computing resources of each Hadoop cluster node. Consequently, this approach leads to excessive CPU utilization. Figure 13-(b) illustrates the CPU usage of Terasort over time. Compared to the native Hadoop framework, the proposed design consumes an average of 1.22× less CPU usage (Figure 13-(b)) and achieves an average of 1.15× faster Terasort execution time (Figure 10-(a)).

Interestingly, as the input data size increases, the RPi 4Bs with a lower parallelism level (i.e., a lower CC count) tend to exhibit better performance, closely associated with the intermediate data that each reduce host writes to disks. The reduce host fetches the output data for each map task over the hyper text transfer protocol (HTTP) and writes them to the local disks [34, 35]. Terasort sorts the input data and does not reduce the final output data size; hence, the output data size for each map task is identical to the corresponding input split size. Thus, the I/O traffic that each reduce host manages is directly proportional to the input data size. Consequently, as the concurrent process count increases, a heavier I/O workload causes a performance bottleneck due to the system overhead from context switching in each JVM.

To investigate the correlation between the input data size and optimal node-level parallelism under I/O-intensive workloads, we prepared another I/O-intensive workload by generating additional heavy I/O traffic under Wordcount. Typically, Wordcount is considered a CPU-intensive workload. However, if a combiner function in the Hadoop MapReduce framework is disabled, Wordcount becomes an I/O-intensive workload. The combiner reduces the volume of data shuffled across the network by aggregating local key-value pairs [23]. Thus, a combiner-disabled Wordcount produces a significant amount of I/O traffic by storing intermediate data (i.e., map output data) on disk and may cause a network bottleneck by transmitting them to each reducer over a network. Table 6 presents the total ex-



(a) Baseline (b) Ours

Figure 14: CPU utilization over time under Grep. Data size: 16 GB, reduce task count: 4.

ecution time of the I/O-intensive Wordcount by CC counts and input data sizes. As the input data size increases from 2 to 32 GB, an optimal parallelism level on the RPi 4B nodes tends to decrease from 5 to 3. This observation is similar to the Terasort experiments (Table 5).

Based on these observations, the proposed Hadoop framework achieves better performance than the original Hadoop under I/O-intensive workloads by effectively reconfiguring node-level parallelism based on the actual computing capability of each SBC node.

4.2.3. Effect of Workloads: CPU-intensive

This section explores the influence of CPU-intensive workloads. The Grep and Wordcount applications count word occurrences from input data and generate fewer final output data [36, 37]. These applications consume considerable CPU computing resources during the map phase, whereas low write I/O traffic is produced to store each map output data. Due to such workload characteristics, improving the map phase performance tremendously affects the overall performance improvement. Thus, determining the relevant number of concurrent tasks has a critical effect.

Theoretically, each JVM process occupies a single CPU core. If the number of processes exceeds the number of CPU cores, the time-sharing OS scheduler triggers frequent system-level interrupts and context switches, affecting CPU utilization. Moreover, if a workload involves high I/O operations, handling I/O exacerbates CPU contention. However, both Grep and Wordcount, unlike Terasort, generates low write I/O traffic using the combiner function. Thus, only a few map output data (i.e., tens of KBs) are stored on the disk. This low write I/O traffic noticeably reduces CPU involvement (i.e., utilization) in I/O management. Therefore, the CPU idle time resulting from the reduced CPU I/O wait time can be employed for more efficient CPU scheduling to accommodate more concurrent processes. Consequently, unlike I/O-intensive workloads, significantly different CPU consumption between our proposed and original Hadoop frameworks was not observed.

Figure 14 illustrates the CPU usage under the Grep workload. The proposed Hadoop framework consumes a lower CPU usage at the user-level by an average of 4% compared with the baseline Hadoop configuration. This

Table 7: Best case performance of the original Hadoop (vs. our redesigned Hadoop framework).

Input data	Terasort		Grep		Wordcount	
	Original	Ours	Original	Ours	Original	Ours
2GB	244.7	157.8	469.8	145.9	504.9	488.7
4GB	420.2	272.5	433.7	177.2	1190.6	605.5
8GB	947.3	495.6	573.9	232.6	2497.3	1115.1
16GB	1473.0	999.0	593.2	356.6	3735.6	2112.3
32GB	2713.8	2153.2	1074.5	604.0	8700.2	3967.0
64GB	5214.6	3679.3	1761.2	1115.0	20592.4	7740.1

outcome is because the baseline Hadoop framework executes an excessive number of concurrent tasks that original Hadoop suggests (i.e., 8 by default). Our redesigned Hadoop framework provides overall performance closer (i.e., 98%) to the optimal performance than the baseline (92%).

Under CPU-intensive workloads, a correlation between the input data size and optimal node-level parallelism is unclear due to the ignorable impact of excessive I/O overhead. Instead, an optimal CC count under CPU-intensive workloads tends to be higher than that under I/O-intensive workloads because of the more effective CPU utilization.

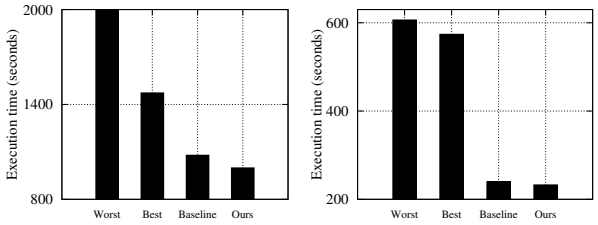
4.2.4. Discussion: Original Hadoop Performance

In Section 3.4, we applied two proposed mechanisms to the original Hadoop framework for the baseline Hadoop framework: (1) adjusting the node-level parallelism of memory-frugal RPi nodes and (2) adding the reduce (or AM) task distribution policy. This section discusses the original Hadoop performance *without* them.

Table 7 presents the total execution time of the original Hadoop framework and our proposed design. Due to the insufficient memory capacity, the original (i.e., unmodified) Hadoop framework cannot run multiple MapReduce tasks on resource-frugal SBC nodes (i.e., RPi 3B and 3B+ with 1 GB of RAM). To resolve this problem, unlike the baseline framework, we added an extra swap space to prevent node failures. However, although the weak node failure problem was solved using the swap space, the original Hadoop framework displays significantly inconsistent performance according to the location of the reduce (or AM) tasks across the heterogeneous SBC Hadoop cluster. Please note that we selected the *best* performance case for the original Hadoop in Table 7.

As in the table, the proposed Hadoop outperforms the original design by an average of 1.53×, 2.19×, and 1.98× under Terasort, Grep, and Wordcount respectively. This result demonstrates that our redesigned Hadoop framework effectively reconfigures node-level parallelism by considering the actual computing power of each RPi node.

Due to the absence of specific reduce (or AM) task assignment policies, the best performance of the original Hadoop occurs when the AM and all reduce tasks are evenly (i.e., not overlapped) distributed to powerful SBC nodes. Similarly, its worst performance is observed when tasks are assigned to weak SBC nodes. Thus, for each execution of an identical MapReduce application, the native Hadoop performance varies from *worst* to *best*.



(a) Terasort (b) Grep

Figure 15: Performance variation of original Hadoop (best vs. worst). Baseline (i.e., slightly modified original Hadoop) performance and our proposed Hadoop performance are provided for reference.

Figure 15 presents the performance variation of the original Hadoop framework between best and worst cases. For reference, the baseline and proposed design performance plots are also provided. Performance of the original Hadoop framework varies greatly according to distribution combinations of both the AM and reduce tasks across cluster nodes (Section 3.1.3). The worst case performance of the original Hadoop is an average of 2.97× and 1.06× slower than the best case performance under Terasort and Grep, respectively. This noticeable performance gap originates from the differing location distribution of the AM and reduce tasks. Therefore, this experiment demonstrates the influence of the redesigned Hadoop YARN scheduling policies. Further, our reconfigurable CC count design guarantees superior performance to even the best case performance of the original Hadoop framework by effectively recalculating the node-level parallelism for each SBC node.

5. Conclusion and Future Work

This paper presented the redesigned Apache Hadoop YARN architecture to improve the application of node-level parallelism on heterogeneous SBC Hadoop clusters. The proposed YARN resource manager locally collects the computational capability of each SBC Hadoop node and automatically determines the appropriate node-level parallelism. When the Hadoop system starts, the proposed Hadoop YARN investigates the computing resources for each SBC node via a newly implemented component (*nodeResources*) in the NodeManager (NM). Then, it automatically calculates the appropriate number of concurrent executable tasks and immediately applies the suggested configurations to each SBC node by overriding the user preconfigurations. The original Hadoop framework cannot change the node-level parallelism until the systems stop even if the configurations are irrelevant for Hadoop applications.

In addition to this automatic node-level parallelism configuration mechanism, the proposed Hadoop framework consistently performs best performance by intelligently distributing reduce (or AM) tasks to an appropriate (not random) cluster node.

Last, this paper explored the influence of workload characteristics and suggested the best configurations by workload. A correlation was observed between the input data size and optimal node-level parallelism (i.e., CC count) under I/O-intensive workloads: as the input data size increases, the optimal CC count tends to decrease. Under CPU-intensive workloads, no correlation was found between the input data size and optimal CC count. Instead, the optimal node-level parallelism tends to be higher than that under I/O-intensive workloads.

The extensive experiments demonstrate that the redesigned Hadoop platform performs better than the baseline Hadoop framework by an average of 15% and 6% under an I/O-intensive workload (Terasort) and CPU-intensive workload (Grep), respectively.

Future work can extend the proposed Hadoop YARN architecture to improve Hadoop MapReduce performance further. This improvement includes a dynamic tuning mechanism with finer-grained MapReduce parameters considering CPU saturation and memory space particularly for low-power computing environments.

6. Acknowledgment

This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the Convergence security core talent training business support program(IITP-2025-RS-2023-00266605) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation).

7. Declarations

- Funding: MSIT(Ministry of Science and ICT), South Korea (IITP-2025-RS-2023-00266605).
- Ethics approval and consent to participate: Not applicable
- Data availability: Not applicable

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. doi: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, Journal of Parallel and Distributed Computing 90-91 (2016) 52–66. doi:https://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 Generation Computer Systems 29 (3) (2013) 739–750. doi:https://doi.org/10.1016/j.future.2012.09.001.

[4] M. Malik, K. Neshatpour, S. Rafatirad, R. V. Joshi, T. Mohsenin, H. Ghasemzadeh, H. Homayoun, Big vs little core for energy-efficient hadoop computing, Journal of Parallel and Distributed Computing 129 (2019) 110–124. doi:10.1016/j.jpdc.2018.02.017.

[5] M. Malik, K. Neshatpour, S. Rafatirad, H. Homayoun, Hadoop workloads characterization for performance and energy efficiency optimizations on microservers, IEEE Transactions on Multi-Scale Computing Systems 4 (3) (2018) 355–368. doi:10.1109/TMSCS.2017.2749228.

[6] W. Wu, W. Lin, C.-H. Hsu, L. He, Energy-efficient hadoop for big data analytics and computing: A systematic review and research insights, Future Generation Computer Systems 86 (2018) 1351–1367. doi:https://doi.org/10.1016/j.future.2017.11.010.

[7] P. P. Nghiem, S. M. Figueira, Towards efficient resource provisioning in mapreduce, Journal of Parallel and Distributed Computing 95 (2016) 29–41. doi:https://doi.org/10.1016/j.jpdc.2016.04.001.
URL https://www.sciencedirect.com/science/article/pii/S0743731516300077

[8] S. Bourhnane, M. R. Abid, K. Zine-dine, N. Elkamoun, D. Benhaddou, Cluster of single-board computers at the edge for smart grids applications, Applied Sciences 11 (22) (2021) 10981. doi:https://doi.org/10.3390/app112210981.

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

[10] 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 Generation Computer Systems 102 (2020) 278–291. doi:https://doi.org/10.1016/j.future.2019.07.040.

[11] S. Bourhnane, M. R. Abid, K. Zine-Dine, N. Elkamoun, D. Benhaddou, High-performance computing: A cost effective and energy efficient approach, Advances in Science, Technology and Engineering Systems Journal 5 (6) (2020) 1598–1608. doi:10.25046/aj0506191.

[12] H. Honar Pajooh, M. A. Rashid, F. Alam, S. Demidenko, Iot big data provenance scheme using blockchain on hadoop ecosystem, Journal of Big Data 8 (2021) 1–26. doi:10.1186/s40537-021-00505-y.

[13] J. Yang, T. Qian, F. Zhang, S. U. Khan, Real-time facial expression recognition based on edge computing, IEEE Access 9 (2021) 76178–76190. doi:10.1109/ACCESS.2021.3082641.

[14] 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. doi:10.1109/W-FiCloud.2016.36.

[15] Z. Guo, G. Fox, Improving MapReduce Performance in Heterogeneous Network Environments and Resource Utilization, in: 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), 2012, pp. 714–716. doi:10.1109/CCGrid.2012.12.

[16] D. Cheng, J. Rao, Y. Guo, X. Zhou, Improving mapreduce performance in heterogeneous environments with adaptive task tuning, in: Proceedings of the 15th International Middleware Conference, 2014, pp. 97–108. doi:10.1145/2663165.2666089.

[17] 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 Generation Computer Systems 89 (2018) 201–212. doi:https://doi.org/10.1016/j.future.2018.06.048.

[18] 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. doi:https://doi.org/10.3390/electronics5030035.

[19] P. Abrahamsson, S. Helmer, N. Phaphoom, L. Nicolodi, N. Preda, L. Miori, M. Angriman, J. Rikkilä, X. Wang, K. Hamily, S. Bugoloni, Affordable and energy-efficient cloud computing clusters: The bolzano raspberry pi cloud cluster experiment, in: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, Vol. 2, 2013, pp. 170–175. doi:10.1109/CloudCom.2013.121.

[20] F. P. Tso, D. R. White, S. Jouet, J. Singer, D. P. Pezaros, The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures, in: 2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops, 2013, pp. 108–112. doi:10.1109/ICDCSW.2013.25.

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

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

[23] T. White, Hadoop: The Definitive Guide, 4th Edition, ÓReilly Media Inc., Sebastopol, CA, USA, 2015.

[24] K. Kc, V. W. Freeh, Dynamically controlling node-level parallelism in hadoop, in: 2015 IEEE 8th International Conference on Cloud Computing, IEEE, 2015, pp. 309–316. doi:10.1109/CLOUD.2015.49.

[25] Z. Li, H. Shen, Measuring scale-up and scale-out hadoop with remote and local file systems and selecting the best platform, IEEE Transactions on Parallel and Distributed Systems 28 (11) (2017) 3201–3214. doi:10.1109/TPDS.2017.2712635.

[26] 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. doi:10.1109/ICSEC.2014.6978206.

[27] 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, Computers and Electrical Engineering 104 (2022) 108403. doi:https://doi.org/10.1016/j.compeleceng.2022.108403.

[28] S. Paik, R. Goswami, D. Sinha Roy, K. H. Reddy, Intelligent data placement in heterogeneous hadoop cluster, 2018, pp. 568–579. doi:10.1007/978-981-10-8657-1_43.

[29] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, X. Qin, Improving mapreduce performance through data placement in heterogeneous hadoop clusters, in: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, pp. 1–9. doi:10.1109/IPDPSW.2010.5470880.

[30] 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 Computing 25 (5) (2022) 3193–3211. doi:https://doi.org/10.1007/s10586-021-03530-x.

[31] N. S. Naik, A. Negi, T. B. B.R., R. Anitha, A data locality based scheduler to enhance mapreduce performance in heterogeneous environments, Future Generation Computer Systems 90 (2019) 423–434. doi:https://doi.org/10.1016/j.future.2018.07.043.

[32] T. T. Htay, S. Phyu, Improving the performance of hadoop mapreduce applications via optimization of concurrent containers per node, in: 2020 IEEE Conference on Computer Applications(ICCA), 2020, pp. 1–5. doi:10.1109/ICCA49400.2020.9022836.

[33] J. P.S., P. Samuel, Analysis and modeling of resource management overhead in hadoop yarn clusters, in: 2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech), 2017, pp. 957–964. doi:10.1109/DASC-PICom-DataCom-CyberSciTec.2017.159.

[34] J. Wang, M. Qiu, B. Guo, Z. Zong, Phase-reconfigurable shuffle optimization for hadoop mapreduce, IEEE Transactions on Cloud Computing 8 (2) (2015) 418–431. doi:10.1109/TCC.2015.2459707.

[35] Y. Guo, J. Rao, D. Cheng, X. Zhou, ishuffle: Improving hadoop performance with shuffle-on-write, IEEE transactions on parallel and distributed systems 28 (6) (2016) 1649–1662.

doi:10.1109/TPDS.2016.2587645.

[36] D. Park, J. Wang, Y.-S. Kee, In-storage computing for hadoop mapreduce framework: Challenges and possibilities, IEEE Transactions on Computers (2016) 1–14doi:10.1109/TC.2016.2595566.

[37] N. Ahmed, A. L. Barczak, T. Susnjak, M. A. Rashid, A comprehensive performance analysis of apache hadoop and apache spark for large scale data sets using hibench, Journal of Big Data 7 (1) (2020) 110. doi:10.1186/s40537-020-00388-5.