ORCA: An Offloading Framework for I/O-Intensive Applications on Clusters

Ji Zhang, Xunfei Jiang, Yun Tian, Xiao Qin
Auburn University
Auburn, AL 36849-5347
{jzz0014,xzj0009,tianyun,xqin}@auburn.edu

Mohammed I. Alghamdi
Al-Baha University
Al-Baha City, Saudi Arabia
mialmushilah@bu.edu.sa

Maen Al Assaf
University of Jordan
Amman, Jordan
m_alassaf@ju.edu.jo

Meikang Qiu
University of Kentucky
Lexington, KY 40506-0046
mqiu@engr.uky.edu

***Abstract*—This paper presents an offloading framework - ORCA - to map I/O-intensive code to a cluster that consists of computing and storage nodes. To reduce data transmission among computing and storage nodes. our offloading framework partitions and schedules CPU-bound and I/O-bound modules to computing nodes and active storage nodes, respectively. From developer's perspective, ORCA helps them to deal with execution-path control, offloading executable code, and data sharing over a network. Powered by the offloading APIs, developers without any I/O offloading or network programming experience are allowed to write new I/O-intensive code running efficiently on clusters.**

We implement the ORCA framework on a cluster to quantitatively evaluate performance improvements offered by our approach. We run five real-world applications on both homogeneous and heterogeneous computing environments. Experimental results show ORCA speeds up the performance of all the five tested applications by a factor of up to 90.1% with an average of 75.5%. Moreover, the results confirm that ORCA reduces network burden imposed by I/O-intensive applications by a factor of anywhere between 35 to 68.

***Keywords*-offloading; I/O intensive;**

I. Introduction

Although offloading techniques have been applied to a wide range of computing platforms (*e.g.* parallel file systems [13] [18] and object-based storage [12]), there is a lack of a general network offloading framework tailored for I/O intensive applications running on clusters. Moreover, none of existing works pay any attention on offloading application development from developers' perspective. Based on our experience, writing an appropriate offloading program is difficult and time-consuming. In this paper, we propose a new offloading framework called ORCA to map I/O-intensive code to a cluster that consists of computing and storage nodes.

Motivations. The following two factors motivate us to develop the ORCA offloading framework:

- heavy network traffic is imposed by transmitting data from storage to computing nodes in clusters, and
- writing an offloading program without any general framework is difficult.

Due to the nature of I/O intensive applications, heavy network traffic is caused by retrieving massive amount of data between computing and storage nodes in clusters. During the data staging phase, data to be processed by applications running on computing nodes must be loaded from storage nodes through interconnections. Transferring huge amount of data can slow down the performance of the applications. This I/O problem becomes even worse for clusters using the Ethernet, where all nodes share network bandwidth in the clusters.

The second motivation driving us is that studies of offloading development are missing. Even for an experienced developer, a number of issues related to the offloading development are difficult to solve, including appropriately designing offloading programs, accurately deciding the I/O-bound modules of the programs, controlling execution paths and efficiently sharing data.

Our goal is to address the above two issues by developing the ORCA framework to automatically offload I/O-bound modules of an application to active storage nodes in a cluster. The offloading framework deals with configurations, execution-path control, offloading executable code, and data sharing. Our framework coupled with an application programming interface (API) and a run-time system enables programmers without any I/O offloading experience to easily write new I/O-intensive code or extend existing code running efficiently on clusters.

Contributions. The main contributions of this work are:

- We describe the ORCA framework centered around an offloading API and a run-time system (see Sec. IV).
- We discuss the implementation details, including the issues of configurations, programming interface, and data sharing (see Sec. V).
- We develop a testbed to evaluate real-world applications in our run-time system (see Sec. VI).
- We present experimental results to show that both homogeneous and heterogeneous clusters powered by ORCA experience reduced amount of network bandwidth used to transfer data among computing and storage nodes (see Sec. VII).

Online resources. The source code and documentation of our I/O offloading framework are freely available at http://www.eng.auburn.edu/~xqin/software/offloading

In the next section, we discusses related work. The main idea behind our offloading framework is highlighted in

Section III. Sections IV and V describe design and implementation details of the framework. Section VI outlines our experimental testbed and methodology. Benchmark applications and performance analysis are presented in Section VII. Conclusions and future plans are discussed in Section. VIII.

II. Related work

The concept of *active disks* was proposed by Acharya *et al.* [9]. In their active disk architecture, processing power and memory are deployed into individual disks, to which a portion of application computation can be offloaded by using a stream-based programming model. Their simulation results show that significant performance improvement can be achieved by reducing data traffic. Riedel *et al.* designed a similar system that moves an application's processing to disk drives. In addition, they developed an analytical model, evaluating a wide range of applications that may benefit from the active storage [19].

Lim *et al.* designed an active-disk-based file system (ADFS), in which application-specific operations can be executed by disk processors [16]. When a file is loaded, only results processed by an application are returned. Moreover, the ADFS file system also offloads a part of file system functionalities (*e.g. lookup*) to active disks. This approach is able to significantly reduce the workload of central file management. Chiu *et al.* presented a distributed architecture that utilizes smart disks equipped with processing power, on-disk memory, and network interfaces [10]. A set of representative I/O-intensive workloads are evaluated on the architecture. Their experimental results suggest that the distributed architecture outperforms partially distributed and centralized systems.

The idea of active storage was implemented in the Lustre file system by Felix *et al.* [13]. Afterwards, piernas *et al* proposed an active storage for Lustre in the user space [18]. Both approaches reduce data movements and improve computing capability. Compared with the kernel-based implementation, the user-space approach is faster, more flexible and readily deployable. In addition, motivated by requirements of specific applications, piernas *et al.* designed and evaluated an efficient way to manage complex striped files in active storage [17].

Du designed an intelligence storage system that combines active disks and object-based storage device (OSD) [12]. Du's study mainly focused on fundamental changes in existing storage systems; he proposed a number of future research directions in the realm of OSD-based storage. Son *et al.* investigated an active storage in the context of parallel file systems [22]. Based on the analysis of parallel applications, they designed an enhanced programming interface that enables application codes to embed in the parallel file system. Moreover, their approach also provides server-to-server communication for reduction and aggregation.

Although the offloading techniques have been extensively explored in the aforementioned studies, our approach differs from the above solutions with respect to the following three issues. First, recognizing that there is a lack of generic offloading framework, we propose a general offloading programming model that can be applied to either sequential or parallel applications (e.g., multi-thread and multi-process programs). We introduce the concept of offloading domains to represent computation consequences. Server-to-server communications proposed by Son *et al.* [22] can be converted into domain-to-domain communications. Second, developing offloading-oriented applications is non-trivial from programmers' perspective. In this study, we address a number of critical implementation issues raised in the development of offloading-oriented programs. These issues, which are crucial to program designs and performance, include I/O-bound module identification, execution path control, and data sharing in an offloading domain. Last, developing offloading-oriented programs in C language is time consuming due to the complexity of the programming language and; therefore, we propose an approach that is able to share dynamic and static data (e.g., codes).

III. I/O Offloading Framework

We will begin this section by highlighting the main idea of our offloading framework for I/O-intensive applications. Then, we will discuss structures of applications designed to gain maximum benefit from the I/O offloading framework.

A. System Architecture

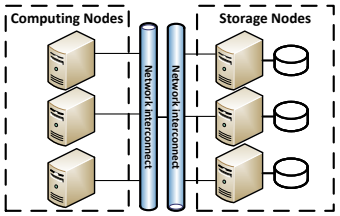


Figure 1. The architecture of commodity clusters, where a number of nodes are connected with each other through interconnects. We focus on clusters enhanced with active storage nodes that have computing capability.

Fig. 1 illustrates the architecture of commodity clusters, where a number of nodes are connected with each other through interconnects. In this work, we focus on clusters enhanced with active storage nodes that have computing capability. In our study, a cluster has two types of nodes: (1) computing nodes that deal with CPU-bound jobs and (2) storage nodes that are responsible for storing data and processing I/O-bound jobs.

In many existing clusters, parallel file systems (see, for example, Lustre [21] and PVFS[14]) are employed to distributed data across multiple storage nodes. To support data-intensive applications, the parallel file systems need to transfer files back and forth between computing and

storage nodes. Although high peak aggregate I/O bandwidth can be achieved by accessing multiple storage nodes in parallel, moving data between computing and storage nodes will inevitably slow down the performance of I/O-intensive applications. Our preliminary evidence shows that reducing the amount of data transferred among nodes is a practical approach to boosting the overall performance of clusters.

B. Structure of Applications in the Offloading Framework

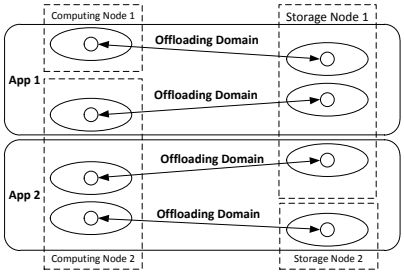


Figure 2. An offloading domain is a logic processing unit, in which a pair of computation and offloading modules are coordinating. I/O-bound modules are assigned to and executed on storage nodes; whereas CPU-bound modules are handled by computing nodes. The framework strives to overlap the executions of CPU-bound and I/O-bound modules to achieve high performance.

Fig. 2 depicts our network offloading framework, in which I/O-bound modules are assigned to and executed on storage nodes. The goal of this framework is to reduce the amount of data transferred from storage nodes to computing nodes. Our offloading idea is inspired by the observation that many I/O-intensive applications (see Section VI for real-world examples) can be partitioned into CPU-bound and I/O-bound modules. CPU-bound modules are handled by computing nodes; whereas I/O-bound modules, running on storage nodes, are referred to as offloading parts. To achieve high performance, the framework makes an effort to overlap the executions of CPU-bound and I/O-bound modules on computing and storage nodes in a cluster.

We now introduce the concept of offloading domains, which are used to group CPU-bound and I/O-bound modules. An offloading domain is a logic processing unit, in which a pair of computation and offloading modules are coordinating. An application may contain either only one offloading or multiple domains. The number of offloading domains in an application heavily depends on the application's design and the number of offloading modules. Offloading domains are independent of, and isolated from each other in the sense that one offloading domain can not be interfered with by the others.

Moreover, two simple applications that create two offloading domains are demonstrated in fig. 2. *App 1* is a multi-process program, in which the CPU-bound modules in both offloading domains are allocated on two computing nodes. The corresponding I/O-bound modules are executed on *storage node 1*. This is a typical n-to-1 model that multiple computer nodes and a storage node are used by *App 1*. On the other hand, *App 2* shows a typical 1-to-n model that a computing node and multiple storage nodes are utilized. The CPU-bound modules of *App 2* can be created as either threads or processes. The I/O-bound modules are offloaded to separated storage nodes. More complex applications can be derived from the two simple examples.

CPU-bound and I/O-bound modules in an offloading domain are, in some cases, serially and synchronously executed. Thus, while CPU-bound modules are running on computing nodes, their I/O-bound counterparts must be waiting and vice versa. In the case where CPU-bound and I/O-bound modules in an offloading domain are asynchronous, our framework can overlap the executions of the CPU-bound and I/O-bound modules on computing and storage nodes to achieve high performance in a cluster.

IV. Design Issues

Before developing the proposed I/O offloading framework, we need to address the following four design issues.

A. Data-intensive module Identification

The first step in partitioning a data-intensive application is to identify the I/O-bound modules of the application. Intuitively, I/O-bound modules need to process huge amount of data, meaning that I/O time should dominate the performance of such modules. On the other hand, CPU-bound modules spend the majority of their time using CPUs to do calculations. A profiling and performance analysis tool can be employed to evaluate whether modules in a data-intensive application are CPU-bound or I/O-bound. With the performance analysis tool in place, programmers can evaluate whether applying the offloading technique improves overall application performance. Such an evaluation process should take into account various aspects such as computing workload, I/O workload, and network traffic.

B. Offloading a program

The second design issue is that of an efficient way of offloading an executable file to an active storage node. Two practical approaches to offloading executable modules are dynamic offloading and static offloading. The main idea of dynamic offloading is to automatically transfer an executable file and its configurations to storage nodes in a cluster before loading the file into the memory. In this method, the offloading platform must be aware of details of the run-time system implementation (e.g., programming languages and libraries) if the run-time system is platform dependent. If the run-time system is platform independent(e.g., implemented in scripts or java), the offloading platform does not have to consider run-time system details. Thus, the level of difficulty in implementing the offloading technique using dynamic distributions highly relies on the nature of the applications to be supported by the framework.

Dynamic offloading introduces another challenge - version management - for platform-dependent applications. In heterogeneous environments, all types of executable files, each of which is dedicated to a specific hardware platform, need to be precompiled. To invoke I/O-bound modules offloaded to storage nodes, applications must detect the type of hardware/software in the storage nodes and choose a proper version of the I/O-bound module to be offloaded on the fly. Moreover, this dynamic-distribution approach suffers from repeatedly transferring I/O-bound modules from computing to storage nodes. Although storage nodes are able to cache and reuse offloaded modules, it is time consuming for computing nodes to decide whether the cached ones on the storage end are valid and updated.

Unlike dynamic offloading, static offloading configures offloaded I/O modules a priori. Static offloading encompasses three distinct procedures if active storage systems are heterogeneous in nature. The first procedure is to manually compile I/O-bound modules for various hardware and runtime systems in heterogeneous storage systems. The second procedure is to write specific configuration files. The last procedure is to deploy the configuration files along with I/O-bound modules onto target storage nodes. Although these three procedures are seemingly complicated, they can be automatically completed by a simple yet efficient tool in our offloading framework. Moreover, the static offloading approach greatly simplifies the design of our offloading framework, because there is no need to address the platform-dependent issues. In this approach, when an application starts, its offloaded I/O-bound modules have already been compiled and installed on storage nodes.

C. Controlling an execution path

The third design issue is a mechanism for transferring executions back and forth between a pair of CPU-bound and I/O-bound modules in an offloading domain.

A feasible option for our framework is Remote Procedure Call (RPC), which is a broadly accepted method of invoking a function to execute in a remote machine. Thanks to RPC's simplicity, it is easy for any programmer to learn and use. There are many RPC libraries implemented by various general-purpose programming languages. RPC was applied to implement Network File System (NFS) [20], MapReduce [11] and Hadoop [8].

D. Data sharing among storage and computing nodes

The last issue is data sharing between a pair of CPU-bound and I/O-bound modules in an offloading domain. Shared data include both global variables and code segments. A major challenge is that in an offloading domain, global variables can not be shared by CPU-bound and I/O-bound modules allocated to different computing and storage nodes.

An intuitive solution for the above challenge is to establish a synchronization mechanism to allow a pair of modules in an offloading domain to notify each other when any global variable is updated. For example, if a CPU-bound module modifies shared data on a computing node, a notification along with the updated data will be delivered to the corresponding I/O-bound module on a storage node.

A second solution is motivated by an observation that in some cases, I/O-bound modules are synchronized with their CPU-bound modules. In a synchronization process, offloaded I/O-bound modules are unable to access global data on storage nodes until control is regained from CPU-bound modules on computing nodes. Thus, CPU-bound modules can transfer updated shared variables to I/O-bound modules by appending the shared variables with offloading requests. In our approach, the framework updates global variables before processing offloading requests. In other words, the changes that occur at offloaded modules can be treated as results in response messages.

Code segments are considered to be a special type of global data. In applications implemented by compiled languages, function objects can not be shared directly. The reason for this is that addresses of an function in a pair of CPU-bound and I/O-bound modules may be different after being loaded into the main memory. On the other hand, in interpreted applications, functions are parsed by names rather than addresses. Hence, both the CPU-bound and I/O-bound modules in an offloading domain are able to obtain identical functions by their names.

In this subsection, we only highlighted the basic idea of data sharing supported in our offloading framework. Please refer to Section V-C for implementation details on the data-sharing mechanism .

V. Implementation Details

In this section, we will describe the implementation details of our offloading framework and explain how to run offloading applications on clusters.

A. Configuration

Recall that we took the static offloading approach (see Section IV-B) by adopting the pre-configuration method to offload I/O-bound modules to storage nodes. The following five steps are required to run a data-intensive application in our offloading framework.

1. Design a data-intensive application and identify I/O-bound modules to be offloaded to storage nodes.
2. Convert the application into its offloading version by using the offloading programming interface (API) described in Section V-B. Developers may need to write configuration files.
3. Create executable files for target storage nodes if the executables are implemented by compiled languages. If the application is developed by interpreted languages, then source files are executable.

4. Copy executable and configuration files to specified directories on computing and storage nodes.
5. Start I/O-bound modules on storage nodes followed by computing nodes. This order is important because offloaded modules must provide services to CPU-bound modules in an initial phase.

B. Offloading API (Application Programming Interface)

The current version of the offloading framework provides an application programming interface (API) for C and C++ languages. Similar APIs can be implemented in other languages like java or python. Our offloading framework provides four API sets summarized in Table I.

The `init` function in the first group initializes and sets up the offloading environments. Programs must execute `init` before issuing any offloading requests. First, `init` decides the role – a CPU-bound or I/O-bound module – that the program plays by identifying a dedicated command-line argument. After the role decision, `init` removes the dedicated argument which cannot longer be accessed. Then, a serial of `MARSHAL` and `UNMARSHAL` functions for primitive data types (*e.g.*, `char` and `unsigned short`) are registered for supporting primitive types serialization.

The second set of function in Table I is to register offloading entries. In C/C++ applications, offloading entries are addresses of functions in offloaded I/O-bound modules. After compilation, all functions are converted into addresses; an identical function may have different addresses in CPU-bound and I/O-bound modules. In order to exchange offloading entries between a pair of CPU-bound and I/O-bound modules, we enable applications to call `register_function` to register functions and then exchange function names instead of addresses. Addresses are automatically converted to names in CPU-bound modules and reverse in offloaded I/O-bound modules by calling `find_name_by_func_addr` and `find_func_by_name` respectively.

Table I Offloading Programming Interface

Interface & Description
`void init ()` Initialize the system.

`void register_function (func_addr)`
build a map from function addresses to their names.
`func_name find_name_by_func_addr (func_addr)`
Get a function name by a given address.
`func_addr find_func_by_name (func_name)`
Get a function address by a given name.

`void MARSHAL (void* obj, char**buf, int* len)`
Serialize an object pointed by obj into a data stream. The address and size of the data stream are specified by buf and len.
`void UNMARSHAL (void* obj, char*buf, int len)`
Un-serialize an object pointed by obj from a data stream. The address and size of the data stream are specified by buf and len.

`void offload_call (addr, func_name, ins, outs)` Invoke an offloading procedure named by func_name. addr indicates a network address (*e.g.*, an IP address) of the target node. The input parameters and results are specified by ins and outs.

The goal of the third API set in Table I is to send and receive parameters and results. Both `MARSHAL` and `UNMARSHAL` accept input parameters *object* in the type of *void* * in order to adapt all types of objects. The following two parameters specify the buffer of the data stream and its length. All data being exchanged between CPU-bound and I/O-bound modules must implement corresponding `MARSHAL` and `UNMARSHAL` functions that are automatically called by the offloading framework. If a function pointer need to be serialized or un-serialized, the pointer has to be converted to the function name by a second set of interfaces and then processed as a regular string. These functions must be registered during initialization as well.

`offload_call` is a real action for calling an offload. The parameter `addr` indicates the network address (*e.g.*, an IP address) of the node where an offloading part will take place. `func_name` specifies an offloading entrance. `ins` and `outs` are an input and output parameters defined as instances of the `offloading_para` structure.

C. Sharing Data

Recall that the complexity of offloading programs heavily depends on data sharing mechanisms (see Section IV-D). Because our goal is to keep offloading programs simple, our framework offers a simple yet efficient way of passing data as input and output parameters. We consider two key issues regarding data sharing.

The first one is how to share global data between computing and storage nodes. All data accessed by both nodes should be overseen by input parameters and results (see Section IV-D), and is required to be deeply copied in `MARSHAL` and `UNMARSHAL` instead of merely copying object points. This is because objects created in address spaces are totally different in the two parts. A function pointer is the data that maintains the address of the function. The address has to be converted to the function name in `MARSHAL` and recovered in `UNMARSHAL`, since the function name keeps consistent in both CPU-bound and I/O-bound modules. The conversion can be completed by Dynamically Loaded (DL) libraries[1] if the function is defined as $extern$. The CPU-bound and I/O bound modules are responsible for handling global updates.

The second issue is how to share code segments. Function entries or executable objects are a special type of data in programs. The framework can not simply copy binary codes and transfer them to another node, because the code might be not executable. In our implementation, we link all object codes to each part, regardless of whether the codes are used or not; therefore, programmers do not need to identify which functions belong to either parts or both. To transfer a function entry, we build a map between function names

[1] http://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html

and addresses, thereby placing function names in offloading requests and responses. Both computing and active storage nodes can resolve function names and addresses by using the offloading API.

VI. Evaluations

A. Experimental Testbed

We set up a homogeneous cluster and a heterogeneous cluster as two testbeds to evaluate real-world applications supported by our offloading framework. Both clusters are comprised of 16 nodes, which form 8 independent offloading domains (see Fig. 2 for an example of offloading domains).

Table II summarizes the hardware and software configurations of the two types of nodes - Type I and Type II - used in our testbeds. Type I nodes have better CPU performance and larger main memory than Type II nodes. Interestingly, measurements collected by hdparm [3] indicate that Type II nodes have higher sequential I/O throughput (130.35 MBytes/Sec.) than Type I nodes (106.94 MBytes/Sec.).

The homogeneous cluster is made up of 16 Type I nodes; the heterogeneous cluster contains 8 Type I nodes and 8 Type II nodes. In the second testbed, computing nodes are Type I and storage nodes are Type II.

B. Benchmark Applications

1) Applications: We tested five benchmarks (see Table III), which are well-known data-intensive applications. PostgreSQL, Word Count(WC), Sort, and Grep were downloaded from their official websites, whereas the Inverted Index application was implemented by our research group at Auburn. In our experiments, we ran the baseline applications on computing nodes and loaded data from the storage nodes through the Network File System (NFS) service [20], that is used to manage massive amount of data in numerous commercial products [4] [15].

We applied the offloading framework to the five benchmark applications, each of which has an I/O-bound module running on storage nodes. In particular, the "executor" is defined as an offloaded module in PostgreSQL. The framework offloads the I/O-bound modules of the other applications to storage nodes. Table III describes the implementation of these benchmarks.

Table II Hardware and Software Configurations

Name	Hardware	Software
Type I	1 × Intel Xeon 2.4 GHz processor 1 × 2 GBytes of RAM 1 × 1 GigaBit Ethernet network card 1 × Seagate 160 GBytes Sata disk [6]	Ubuntu 10.04 Linux kernel 2.6.23
Type II	1 × Intel Celeron 2.2 GHz processor 1 × 1 GBytes of RAM 1 × 1 GigaBit Ethernet network card 1 × WD 500 GBytes Sata disk [7]	Ubuntu 10.04 Linux kernel 2.6.23

2) Data Preparation: To measure performance of PostgreSQL running in our offloading framework, we created four databases with sizes of 5 GBytes, 10 GBytes, 15 GBytes and 20 GBytes. No index was generated in these databases; therefore, PostgreSQL had to directly access data in the tables rather than merely checking index structures during query processing. Each database is made up of 1,000 tables, each of which has 100 integer attributes. Tuples are equally distributed across these tables, so a larger database has more tuples in each table. We generated 1,000 queries, each of which scans only one table. Together, these queries cover all the tables in the database.

For the other four benchmark applications, we created five text files of relatively smaller sizes (i.e., 400 MBytes, 600 MBytes, 800 MBytes, 2 GBytes and 4 GBytes). Each text file contained a number of randomly generated words. Due to the limitation of the main memory, we tested the inverted index application using the first three text files on the homogeneous cluster. This was because frequent page faults made I/O noise in the experiments when the input file size was larger than the main memory. We also tested the other four applications on the heterogeneous cluster.

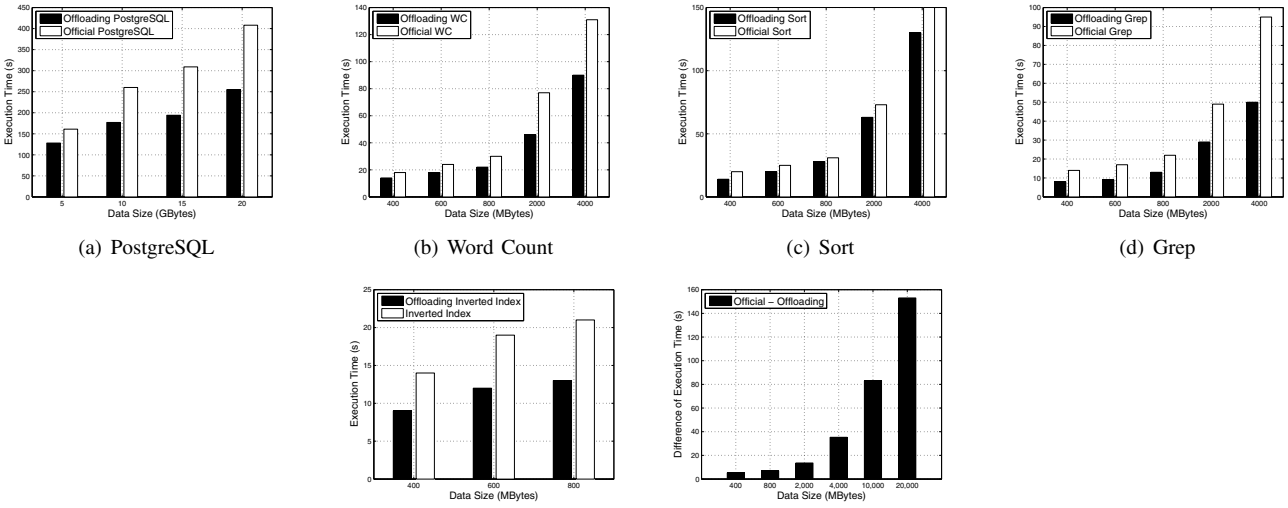
VII. Results

A. Overall Performance Evaluation

1) Homogeneous Clusters: Fig. 3 illustrates the execution times of the five applications (see Table III) to compare the ORCA-enabled cluster against the same cluster without I/O offloading. The results show that the offloading framework significantly reduces the execution times of all five tested applications. For example, when data size is 4 GBytes, our

Table III Real-World Benchmark Applications

Applications	Descriptions
PostgreSQL 9.0 [5]	It is a relational database management system. The offloading framework offloads the "executor" module to storage nodes. The I/O-bound module receives an execution plan and performs queries. The CPU-bound module manages connections to clients, converts SQL statements to execution plans and sends results back to clients.
Word Count in GNU coreutils 7.4 [1]	It counts the number of words in a set of files. Our framework partitions the Word Count application into an I/O-bound module that calculates word occurrences in one file, and a CPU-bound module that sums the occurrences up.
Sort in GNU coreutils 7.4 [1]	It sorts lines of a text file in alphabetical order. Our framework treats the entire Sort application as an offloaded I/O-bound module that receives a file name and stores sorted text in a file.
GNU Grep 2.7 [2]	It searches through a file for lines which contains a given keyword. The I/O-bound module in Grep finds desired lines in a file; the CPU-bound module in Grep transfers keywords and file names to the I/O-bound module.
Inverted Index (our benchmark)	It loads a set of files and builds a map between words to their occurrences. In the Inverted Index application, its I/O-bound modules constructs a map for each file; a CPU-bound module transfers file names to the I/O-bound module.



(a) PostgreSQL (b) Word Count (c) Sort (d) Grep

(e) Inverted Index (f) Performance Improvement

Figure 3. Execution times of the five real-world benchmark applications running on the homogeneous cluster (i.e., the first testbed).

scheme can improve performance of PostgreSQL, and Grep by a factor of 60.8% and 90.1%, respectively.

The applications without the I/O-offloading support are slowed down by remotely accessing huge amount of data, because the data must be transferred from storage nodes to computing nodes. Our framework solves this performance problem by offloading I/O-bound modules to storage nodes, thereby substantially reducing I/O time through local data accesses. Although the applications running in our framework have to exchange input/output parameters among computing and storage nodes, the data size of input/output parameters is significantly smaller than the dataset size.

Fig. 3(f) shows the impact of data size on performance improvement gained by our offloading framework. We measured the performance of the four applications (i.e., WC, Sort, Grep and Inverted Index) on four datasets (400MB, 800MB, 2GB, and 4GB) and PostgreSQL on two large datasets (10GB and 20GB). We plotted performance improvement in terms of execution-time reduction. This reveals that the performance improvements achieved by our offloading framework become more pronounced as the datasets grow in size. When data size is small, the non-ORCA-enabled applications can take advantage of continuous I/O operations optimized by the NFS service. For example, NFS can cache entire datasets in the main memory so that the datasets can be repeatedly processed without further remote I/O accesses. Unfortunately, when the datasets grow in size, the non-ORCA-enabled applications benefit very little from caching due to limited caching ability in computing nodes.

2) Heterogeneous Clusters: Fig. 4 shows execution times of the five benchmark applications supported by the offloading framework on a heterogeneous cluster, in which computing nodes and storage nodes have different performance. The results plotted in Fig. 4 are consistent with those shown in Fig. 3.

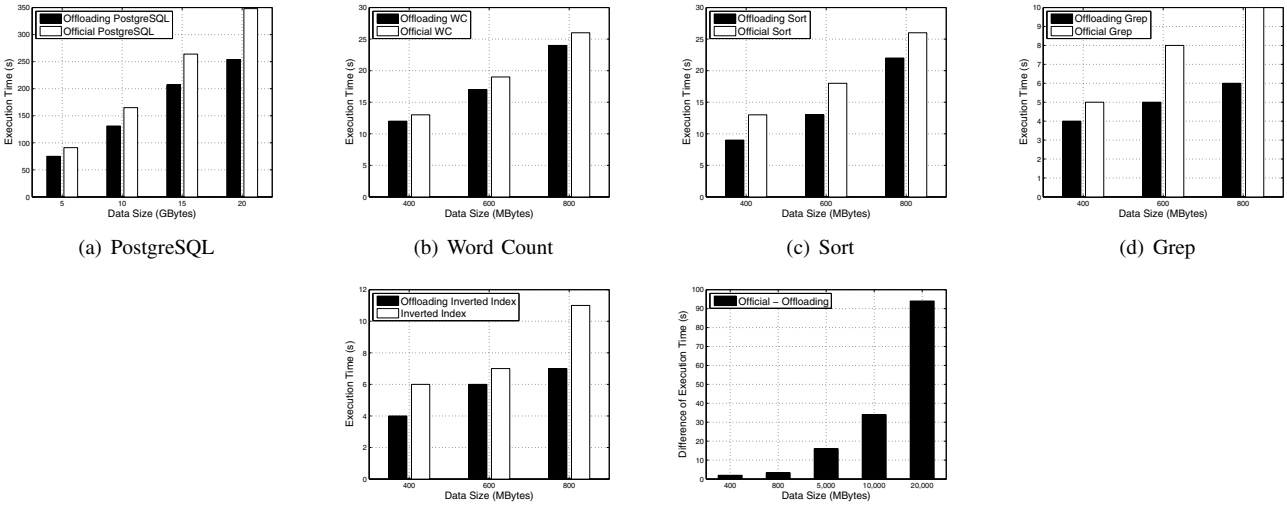
Comparing Figs. 3 and 4, the heterogeneous cluster offers better performance than the homogeneous one. This is because the bottleneck of I/O-intensive applications is the I/O bandwidth, and the storage nodes (i.e., Type II nodes) in the heterogeneous cluster have higher I/O bandwidth than the storage nodes (i.e., Type I nodes) in the homogeneous cluster. Although Type I nodes are superior to Type II nodes in terms of CPU speed and memory capacity, higher I/O throughput of Type II nodes cause the heterogeneous cluster to outperform its homogeneous counterpart.

B. Network Load Evaluation

1) Homogeneous Clusters: Fig. 5 shows network load caused by PostgreSQL when data size is set to 5GB, 10GB, 15GB, and 20GB, respectively. The results confirm that the ORCA framework minimizes the network traffic of the homogeneous cluster running PostgreSQL. For example, ORCA reduces network burden by a factor of anywhere between 35 to 55. When the non-ORCA-enabled PostgreSQL is running, transferring data from the storage to computing nodes keeps the network resources very busy.

Fig. 6 shows network load imposed by the other four applications processing an 800MB dataset. WC, Grep, and Inverted Index share a similar network traffic pattern with PostgreSQL. ORCA reduces network traffic in the cluater by a factor of anywhere between 35 to 45. Fig. 6(b) shows that the data transmission rate in Sort is constantly changing between 0 and 65MB/s.

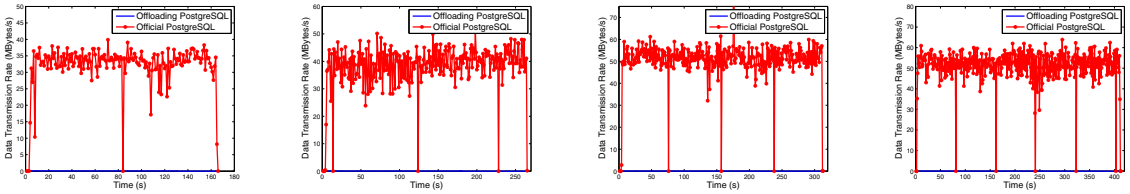
2) Heterogeneous Clusters: Figs. 7 and 8 show the network traffic patterns of the five applications running on the heterogeneous cluster. The empirical results indicate that ORCA lowers network load of the heterogeneous clusters by a factor of anywhere between 35 to 68.



(a) PostgreSQL (b) Word Count (c) Sort (d) Grep

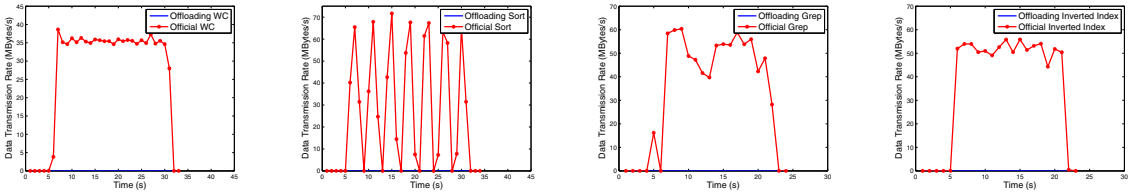
(e) Inverted Index (f) Performance Improvement

Figure 4. Execution times of the five real-world benchmark applications running on the heterogeneous cluster (i.e., the second testbed).



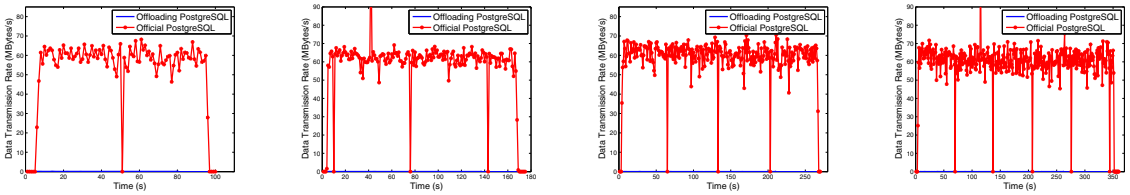
(a) 5 GBytes (b) 10 GBytes (c) 15 GBytes (d) 20 GBytes

Figure 5. Network load imposed by PostgreSQL accessing different databases on the homogeneous cluster (i.e., the first testbed).



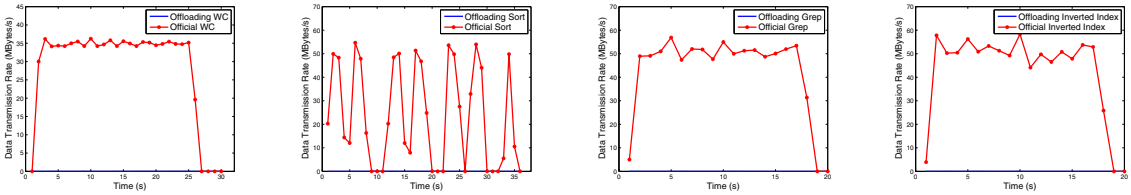
(a) Word Count (b) Sort (c) Grep (d) Inverted Index

Figure 6. Network load imposed by the four real-world applications accessing 800 MB datasets on the homogeneous cluster (i.e., the first testbed).



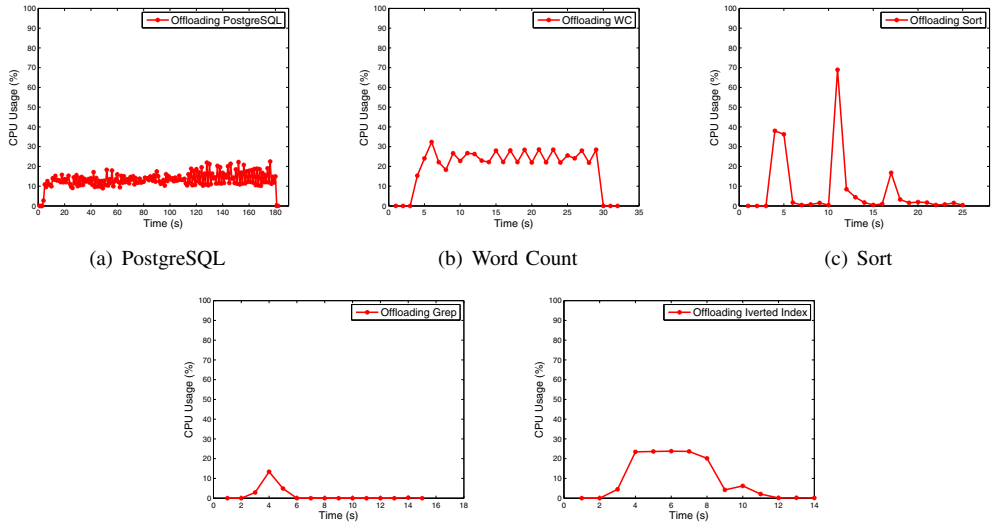
(a) 5 GBytes (b) 10 GBytes (c) 15 GBytes (d) 20 GBytes

Figure 7. Network load imposed by PostgreSQL accessing different databases on the heterogeneous cluster (i.e., the second testbed).



(a) Word Count (b) Sort (c) Grep (d) Inverted Index

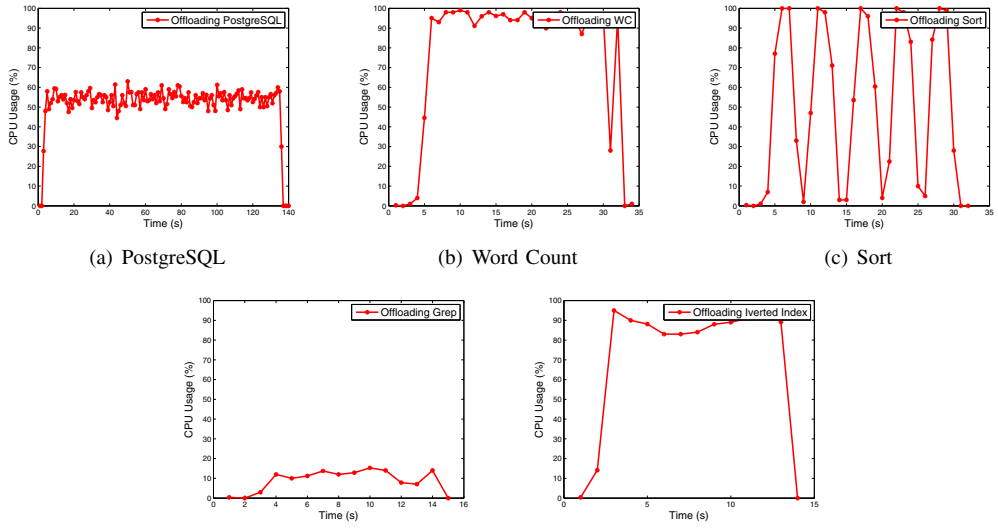
Figure 8. Network load imposed by the four real-world applications accessing the 800 MB datasets on the heterogeneous cluster (i.e., the second testbed).



(a) PostgreSQL (b) Word Count (c) Sort

(d) Grep (e) Inverted Index

Figure 9. CPU load imposed by the five real-world applications in the storage nodes of the homogeneous cluster (i.e., the first testbed).



(a) PostgreSQL (b) Word Count (c) Sort

(d) Grep (e) Inverted Index

Figure 10. CPU load imposed by the five real-world applications in the storage nodes of the heterogeneous cluster (i.e., the second testbed).

C. CPU Usage Evaluation

We observed that the data transmission rate (ranging from 50MB/s to 70MB/s) of the non-ORCA-enabled PostgreSQL on the heterogeneous cluster is constantly higher than that of the homogeneous cluster. In addition, the network links of the homogeneous and heterogeneous clusters are not saturated, because the data transmission rates are below the maximum network bandwidth (i.e., 1 Gbps) in both cases. Data retrieved from the storage nodes can be delivered to the computing nodes; thus, accessing data in the heterogeneous cluster is faster than in its homogeneous counterpart.

1) Homogeneous Clusters: The goal of these experiments was to assess the performance impact of offloaded I/O-bound modules on storage nodes. This goal was achieved by evaluating CPU usage of storage nodes in the homogeneous cluster running the five data-intensive applications. Evaluating CPU usage of storage nodes is very important, because offloaded I/O-bound modules may have side effect on other I/O services running on the storage nodes.

Fig. 9 illustrates CPU utilization of PostgreSQL processing a 10 GB dataset and the other applications processing an 800 MB dataset. We observed that CPU usage, in most cases, is below 30% although there were two cases where the CPU utilization reaches 40% and 70% for a few seconds (see Fig. 9(c)). These two cases have little negative impact on storage nodes. Of all the five tested applications, Grep (see Fig. 9(d)) had the least overall impact on other services running on storage nodes. Overall, we concluded that our offloading framework has minimal negative impact on any

services running on storage nodes in homogeneous clusters.

We confirmed that improving performance of data-intensive applications comes at the cost of increasing CPU usage in storage nodes. Fig. 9 indicates that different offloaded I/O-bound modules lead to different CPU-usage increases in storage nodes. An increase in CPU utilization of storage nodes heavily relies on the complexity of the I/O-bound modules, varying from a simple word counter to a complicated procedure, scanning an entire table.

2) Heterogeneous Clusters: To the heterogeneous clusters, fig. 10 shows CPU usages of storage nodes running the offloaded modules for the five benchmark applications. The results suggest that WC and Inverted Index give rise to a high CPU usage (i.e., $>$90%). The Sort application repeatedly pushes the CPU usage up to 100% and then drops down to nearly 0%. PostgreSQL and Grep keep CPU usage at a moderate level (i.e., 50%-60%) and a low level (i.e., $<$18%), respectively. If storage nodes have low-performance CPUs or offloaded modules are CPU-intensive, then the offloaded modules can cause high CPU utilization in the storage nodes.

VIII. Conclusion and future work

In this study, we proposed the ORCA programming framework to automatically offload I/O-bound modules of applications to storage nodes in a cluster. ORCA aims to reduce network traffic incurred by transferring data among computing and storage nodes in a cluster. The ORCA framework allows programmers to easily write new I/O-bound modules or partition existing code to run efficiently on clusters without imposing heavy network load. Our empirical results show that ORCA achieves two important objectives in both homogeneous and heterogeneous clusters.

For future research, our model can be extended to a multi-offloading-domain model in which multiple offloading domains can be properly coordinated. In light of this new model, we will upgrade the offloading management in our framework. We plan to implement a dispatch manager that allocates I/O-bound modules to appropriate storage nodes.

Acknowledgment

The work in this paper was supported by the US National Science Foundation under Grants CCF-0845257(CAREER), CNS-0757778 (CSR), CCF-0742187 (CPA), and CNS-0917137 (CSR).

References

[1] Gnu core utilities. http://www.gnu.org/software/coreutils/.

[2] Gnu grep. http://www.gnu.org/software/grep/.

[3] hdparm. http://en.wikipedia.org/wiki/Hdparm.

[4] Oracle 11g release 1 rac on linux using nfs. http://www.oracle-base.com/articles/11g/OracleDB11gR1RACInstallationOnLinuxUsingNFS.php.

[5] Postgresql. http://www.postgresql.org/.

[6] Seagate product manual of barracuda 7200.12 serial ata. http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%207200.12/100529369b.pdf.

[7] Wd5000aaks specification. http://www.wdc.com/en/products/products.aspx?id=110.

[8] Apache hadoop. http://lucene.apache.org/hadoop/, 2006.

[9] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, Oct. 1998.

[10] S. Chiu, W.-k. Liao, and A. Choudhary. Design and evaluation of distributed smart disk architecture for i/o-intensive workloads. ICCS'03, pages 230–241, Berlin, Heidelberg, 2003. Springer-Verlag.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[12] D. H. C. Du. Intelligent storage for information retrieval. NWESP '05, pages 214–, Washington, DC, USA, 2005. IEEE Computer Society.

[13] F. E.J., F. K., R. K., and N. J. Active Storage Processing in a Parallel File System. *In Proc. of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*, 2006.

[14] I. F. Haddad. Pvfs: A parallel virtual file system for linux clusters. *Linux J.*, 2000(80es), Nov. 2000.

[15] R. Kassick, F. Boito, and P. Navaux. Impact of i/o coordination on a nfs-based parallel file system with dynamic reconfiguration. pages 199–206, oct. 2010.

[16] H. Lim, V. Kapoor, C. Wighe, and D. H.-C. Du. Active disk file system: A distributed scalable file system. MSS '01, page 101. IEEE Computer Society, 2001.

[17] J. Piernas and J. Nieplocha. Efficient management of complex striped files in active storage. Euro-Par '08, pages 676–685, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] J. Piernas, J. Nieplocha, and E. J. Felix. Evaluation of active storage strategies for the lustre parallel file system. SC '07, pages 28:1–28:10, New York, NY, USA, 2007. ACM.

[19] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. VLDB '98, pages 62–73. Morgan Kaufmann Publishers Inc., 1998.

[20] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Innovations in internetworking. chapter Design and implementation of the Sun network filesystem, pages 379–390. Artech House, Inc., Norwood, MA, USA, 1988.

[21] P. Schwan. Lustre: Building a file system for 1000-node clusters. *In Proceedings of the 2003 Linux Symposium*, 2003.

[22] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W.-K. Liao, and A. Choudhary. Enabling active storage on parallel i/o software stacks. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1 –12, may 2010.