Hadoop Extensions for Distributed Computing on Reconfigurable Active SSD Clusters

ABDULRAHMAN KAITOUA and HAZEM HAJJ, American University of Beirut
MAZEN A. R. SAGHIR, Texas A&M University at Qatar
HASSAN ARTAIL, HAITHAM AKKARY, MARIETTE AWAD, MAGEDA SHARAFEDDINE, and KHALEEL MERSHAD, American University of Beirut

In this article, we propose new extensions to Hadoop to enable clusters of reconfigurable active solid-state drives (RASSDs) to process streaming data from SSDs using FPGAs. We also develop an analytical model to estimate the performance of RASSD clusters running under Hadoop. Using the Hadoop RASSD platform and network simulators, we validate our design and demonstrate its impact on performance for different workloads taken from Stanford's Phoenix MapReduce project. Our results show that for a hardware acceleration factor of 20×, compute-intensive workloads processing 153MB of data can run up to 11× faster than a standard Hadoop cluster.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Reconfigurable Computing, RASSD, Hadoop

Additional Key Words and Phrases: Data-intensive computing, middleware, active storage

ACM Reference Format:
Abdulrahman Kaitoua, Hazem Hajj, Mazen A. R. Saghir, Hassan Artail, Haitham Akkary, Mariette Awad, Mageda Sharafeddine, and Khaleel Mershad. 2014. Hadoop extensions for distributed computing on reconfigurable active SSD clusters. ACM Trans. Architec. Code Optim. 11, 2, Article 22 (June 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2608199

1. INTRODUCTION

The volume of data generated around the globe is increasing at an annual rate of 40% to 60% [Manyika 2011]. This exponential growth is due to increased Internet activities (e.g., search, social networking, media sharing, email, etc.), office automation, and the widespread use of networked, mobile computing devices. Much of this data, commonly called Big Data, is very large and unstructured and often carries a wealth of information that can be used by businesses and organizations to improve decision making, minimize operational risks, and respond more quickly to customer sentiments. Extracting useful information from Big Data is called data analytics, and it requires powerful computing resources that are increasingly moving to the cloud.

This work was made possible by NPRP grant # 09-155-2-066 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.
Author's addresses: A. Kaitoua, H. Hajj, H. Artail, H. Akkary, M. Awad, M. Sharafeddine, and K. Mershad, Electrical and Computer Engineering Department, American University of Beirut, Bless street, Beirut, Lebanon; email: {aak69, hh63, ha27, ha95, ma162, mas117, kwm03}@aub.edu.lb; M. A. R. Saghir, Electrical and Computer Engineering Department, Texas A&M University at Qatar, P.O. Box 23874, Doha, Qatar; email: mazen.saghir@qatar.tamu.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 1544-3566/2014/06-ART22 $15.00
DOI: http://dx.doi.org/10.1145/2608199

Apache Hadoop (Hadoop) is an open-source framework for cloud-based distributed computing that is widely used for data analytics. It consists of two major components: a Hadoop Distributed File System (HDFS), which partitions large datasets into smaller subsets and distributes and replicates them across clusters of commodity computers, and a MapReduce programming model, which uses data-transforming functions called *mappers* to process the partitioned and replicated data subsets in a distributed manner. The results of the mappers are then shuffled, sorted, and finally combined into a unified set of results by data aggregation functions called *reducers*. The two main advantages of Hadoop are scalability and reliability. Hadoop can easily distribute and process data across thousands of computing nodes, resulting in very high computational throughput rates. Constant runtime monitoring and data replication also ensure that Hadoop can quickly recover from hardware or software failures. However, scalability and reliability come at a very high cost. Large amounts of computing, data storage, and networking equipment must be provisioned.

Initially, Hadoop and its MapReduce programming model were designed to run on general-purpose computers and were not aimed at custom hardware. In order to mitigate the costs of operating Hadoop clusters and reduce their physical footprint, processing could be offloaded from general-purpose processors to specialized processors. Graphics processing units (GPUs) are an example of such specialized processors that are currently being used in commercial cloud computing platforms such as Amazon EC2 [xxxx], Nimbix [2014], and Peer1 Hosting [2014]. A GPU contains hundreds of cores that are ideal for running code that exhibits high degrees of data-level parallelism. Recent studies have shown that GPUs can greatly reduce the size of Hadoop clusters while significantly improving performance [Fang 2011; Abbasi 2012].

Field-programmable gate arrays (FPGAs) are another form of specialized processors that can be used in Hadoop clusters. FPGAs are programmable logic devices that can leverage high logic gate densities to exploit parallelism and implement custom hardware accelerators. In fact, recent studies have demonstrated the feasibility of using FPGAs to accelerate mapper and reducer functions [Yeung 2008; Shan 2010]. However, to the best of our knowledge, there has been no prior work on extending the Hadoop framework to support FPGA devices.

In this article, we address this problem by introducing Hadoop extensions to a platform of distributed reconfigurable active solid-state drives (RASSDs) [Abbani 2011]. The distributed RASSD system provides two main features to improve performance. First, the FPGA node allows faster, more efficient computations on reconfigurable hardware accelerators specifically customized to the computational demands of applications. Second, by integrating the SSD storage with the FPGA computing hardware on the same node, the time it takes to transfer data from storage to where it is processed is minimized.

In other words, the RASSD is an active storage device [Keeton 1998; Acharya 1998; Riedel 2001] consisting of a tightly coupled FPGA–SSD pair. The FPGA is configured at runtime to implement an application-specific hardware accelerator that processes streams of data from the SSD. Clusters of RASSD nodes can then be used to process data in a distributed manner. The RASSD platform also uses middleware software to manage the configuration of RASSD nodes and distribute computational tasks across the RASSD clusters. Although our extensions enable these RASSD clusters to operate under the Hadoop framework, the work can be extended to support GPUs clusters as well.

1.1. Article Contributions

We make four key contributions in this article:

1. We propose a set of Hadoop extensions that support data distribution and execution of mapper and reducer functions on FPGA-based RASSD nodes.

2. We present experimental results demonstrating the performance and energy-efficiency advantages of an RASSD node prototype with hardware accelerators using K-means clustering as a case study.
3. We develop an analytical performance model to evaluate the impact of the proposed extensions.
4. We present the results of a quantitative study on the performance of benchmarks from the Stanford Phoenix MapReduce project when executing on RASSD nodes under our proposed extensions.

The rest of the article is organized as follows. Section 2 includes related work, and Section 3 provides an overview of the RASSD platform. Section 4 describes the architecture of an RASSD node prototype and compares its performance and energy characteristics to a commodity PC when executing a K-means clustering workload. In Section 5, we describe the Hadoop extensions and develop our analytical performance model. In Section 6, we use our model to study the performance of Hadoop benchmarks from the Phoenix suite when they are executed on RASSD nodes under our proposed extensions. Finally, we present our conclusions in Section 7.

2. RELATED WORK

This section discusses prior research on active storage and Hadoop extensions.

2.1. Active Storage

Moving data processing functions closer to the storage device has its origins in the database machines of the 1970s and 1980s [Hsiao 1979; Ozkarahan 1975; Su 1979]. It was studied briefly in the late 1990s and early 2000s when active disk drives were introduced. These devices consisted of hard disk drives tightly coupled with local disk processors [Acharya 1998; Gibson 1998; Keeton 1998; Riedel 1999]. In these systems, the computationally intensive portion of an application is called a *kernel*. The higher-level tasks of downloading and initiating the kernels execution on the disk processor and collecting results are handled by a host processor using a streaming programming model [Acharya 1998]. In our work, these application kernels are candidates for hardware acceleration. Furthermore, drivelets are pieces of code running on the RASSD node that initialize and launch the corresponding hardware accelerator.

Active storage architectures require operating system support. Prior work reported in Acharya [1998] developed the DiskOS to support memory management, drivelet scheduling, and stream communication with a host computer. In contrast, our platform supports thousands of distributed reconfigurable active SSD nodes. The complexity of the services required in our reconfigurable RASSD node and within our distributed platform middleware layer has made it necessary to adopt an embedded Linux node operating system [Mendon 2012] and a specialized middleware [Jomaa 2013] that hides the complexity of the RASSD hardware. In this work, we propose new middleware that extends Hadoop, configures the RASSD node, and finds and processes the distributed data using an API abstraction layer. This helps programmers to focus on the application business logic.

FPGAs have also been used to accelerate database operations. IBM/Netezza and XtremeData are two companies developing data warehouse appliances (DWAs). In these systems, the FPGA-enabled, distributed, active disks accelerate key SQL operations (e.g., sorts, joins, and orderings) [Helmreich 2006; Scofield 2010]. Users interact with DWAs through a standard SQL interface, and performance can be scaled by distributing a database across multiple DWA machines. Unlike IBM/Netezza, our RASSD platform is designed for data analytics, and this article extends the Hadoop MapReduce platform to work with RASSD clusters.

2.2. Hadoop

Hadoop is a framework that enables users to distribute data processing among different compute nodes in a system and then aggregate and reduce results. Hadoop consists of three major components: Hadoop Common, which is a set of common libraries and utilities; Hadoop Distributed File System (HDFS), which defines one NameNode at the master server as a metadata manager for a specific cluster of slave nodes, and several DataNodes on which data is stored; and MapReduce programming framework, which schedules computations across slave nodes and aggregates intermediate values to produce the final result [Dean 2008].

The MapReduce framework infrastructure consists of a JobTracker and TaskTrackers. A JobTracker runs on a master node to manage and schedule the map and reduce tasks over the compute nodes, while Task Trackers receive their allocated tasks from the job tracker and manage the execution of tasks. Task trackers communicate with the Job Tracker by sending periodic heartbeats to inform Job Trackers about the current progress of the tasks running and the location of the output files generated by task execution.

Hadoop is becoming increasingly popular with several ongoing projects implemented using the Hadoop core capabilities (i.e., HDFS and MapReduce framework). These include Pig [Apache 2012], which is a high-level data-flow language and execution framework for parallel computation; HBase [2014], which is a distributed database that supports structured data storage for large tables, and Mahout [2014], which is a machine-learning and data-mining library. Several versions of Hadoop have been built to overcome some of Hadoop's shortcomings. For example, most data-mining applications (e.g., neural networks or K-means clustering) involve iterative computations over datasets. A direct Hadoop implementation wastes bandwidth, I/O, and CPU cycles when compared with an optimal execution for a consecutive set of iterations [Chu 2006]. To overcome such issues, Iterative Hadoop (IHadoop) was introduced to handle iterative applications [Elnikety 2011]. Another variant of Hadoop is Phoenix, which includes a MapReduce runtime for shared-memory multicores and multiprocessors [Yoo 2009], adapted for multicore processors with large shared memory.

In Shan [2010], the authors propose a framework for the MapReduce programming model on FPGAs. The work in Shan [2010] differs fundamentally from our scheme in that it does not propose a whole system, but rather aims at comparing hardware to software performance for the MapReduce framework.

The authors in Attebury [2009] adapt Hadoop DFS to build a framework for the Grid Storage Element. They use additional software on top of Hadoop to make the complete file system local to the worker nodes and to enable the transfer of data between storage elements or between storage elements and worker nodes. Another application is the Web Geographic Information System prototype (WebGIS) [Liu 2009], which takes advantage of the Hadoop framework using very small files, while Hadoop's DFS achieves the best performance when processing large files under the default block size. The authors propose a solution that combines the small files into larger ones and keeps an index for each large file. Both of these approaches do not make use of the MapReduce paradigm, which is a major component of Hadoop and a main contributor to its performance. By contrast, our framework completely adapts Hadoop, with its two major components: the DFS and the MapReduce model. Additionally, in this work, we not only benefit from the ability to reconfigure FPGAs but also extend the benefit to use SSD storage that is tightly coupled with the FPGA hardware.

To the best of our knowledge, the closest work to our own is presented in Yeung [2008] and Shan [2010]. The authors in Yeung [2008] describe a MapReduce library that contains functionalities needed to interface to FPGAs and present benchmarks

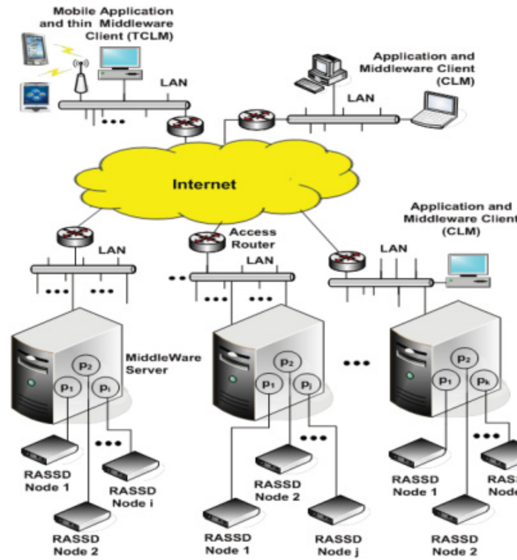

Fig. 1. High-level distributed RASSD system architecture.

to test the performance of running MapReduce functions on custom hardware. Unlike our proposed platform, FPGAs have no ability to interact with the hard disks directly, which forces them to perform this interaction through the CPU. This results in high overhead and represents a bottleneck for data-intensive applications. Moreover, they restricted the work to parallelism on a single FPGA attached to a single host with no Hadoop as a middleware, by running multiple map functions on it, whereas we are proposing a distributed execution model where multiple hosts are attached to multiple FPGAs that run the map functions simultaneously for better performance.

3. DISTRIBUTED RASSD SYSTEM ARCHITECTURE

Figure 1 shows the distributed RASSD system architecture. The system consists of an application client to interface with the middleware (CLM), middleware servers (MWS), and RASSD nodes. All components are interconnected via WANs and LANs. The Slave Middleware Servers (SMWSs) are connected to the RASSD nodes via a LAN. Each RASSD node consists of one Xilinx FPGA board connected to one or more SSD devices using SATA interconnects. Workstation, laptop or handheld computers can be used as clients to run different data processing queries.

In this article, we extend the Hadoop framework to support distributed operations, where an Application/Middleware client is expected to communicate with all SMWSs through a Client Middleware interface (CLM) that communicates with the Master Middleware Server (MMWS). When a CLM sends a processing request to an MMWS, the latter resolves the location of the required data either locally or remotely by communicating with a designated directory service. The CLM also submits to the MMWS a kernel identifier along with the request. In our system, kernels refer to the data processing functions delegated to an RASSD node. In this communication process, the SMWSs send the appropriate kernel function and its hardware accelerator configuration to the RASSD node. Hardware accelerators exploit the reconfigurable FPGA logic fabric to customize computations and achieve significant speedups of the data intensives portions of our applications over software implementations. We emphasize here that we envision future RASSD platforms that do not necessarily require the

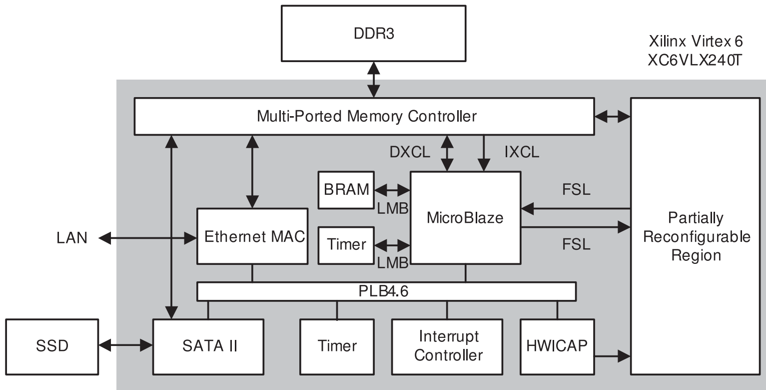

Fig. 2. RASSD node prototype.

application programmer to deal with or know the details of the hardware accelerators and their configuration bitstreams. Instead, the programmer writes applications using the extended MapReduce model Java functions. We describe the implementation of this middleware functionality in detail in Section 5.

4. RASSD NODE PROTOTYPE

In this section, we describe the design of an RASSD node prototype and compare its performance and energy characteristics to a commodity personal computer similar to those used in contemporary Hadoop clusters. Figure 2 is a block diagram of our RASSD node prototype, which we implemented in a Xilinx XC6VLX240T Virtex 6 FPGA found on the ML605 development board. We also used a 60GB OCZ Vertex Plus R2 SSD, which we connected to the ML605 board through a SATA II port on a Xilinx FMC XM104 connectivity card.

The FPGA implements a system-on-chip consisting of a MicroBlaze soft processor core, various peripheral controllers, and a partial reconfigurable region (PRR). The PRR is a user-defined region of the FPGA logic fabric that hosts hardware accelerators. The MicroBlaze processor runs a lightweight monitor program (RASSD OS) [Ali 2012] based on embedded Linux. It performs supervisory functions such as communicating with middleware servers, initiating data transfers between the Ethernet MAC or SSD and DDR3 memory, and loading hardware configuration bitstreams into the PRR. The MicroBlaze processor also runs accelerator driver codes called drivelets that enable it to communicate with a hardware accelerator in the PRR through a pair of fast simplex links (FSLs). The FSLs initialize or read status information from a hardware accelerator. The MicroBlaze processor operates at 150MHz.

The RASSD node also includes a multiported memory controller (MPMC) that provides the MicroBlaze processor, the PRR, the Ethernet MAC, and the SATA II controller with access to the external, 512MB, DDR3 memory. The PRR is connected to the MPMC through a 64-bit native port interface (NPI) personality interface module (PIM) operating at 200MHz. Other peripheral controllers in the RASSD node include a PLB timer to support multitasking in RASSD OS, an interrupt controller, and an HWICAP controller to configure the PRR with partial bitstreams for hardware accelerators.

4.1. K-Means Hardware Acceleration

To validate the RASSD node prototype, we developed a set of hardware accelerators for the K-means clustering algorithm. K-means clustering partitions a set of n samples in an m-dimensional space into k clusters based on the minimum Euclidean distance

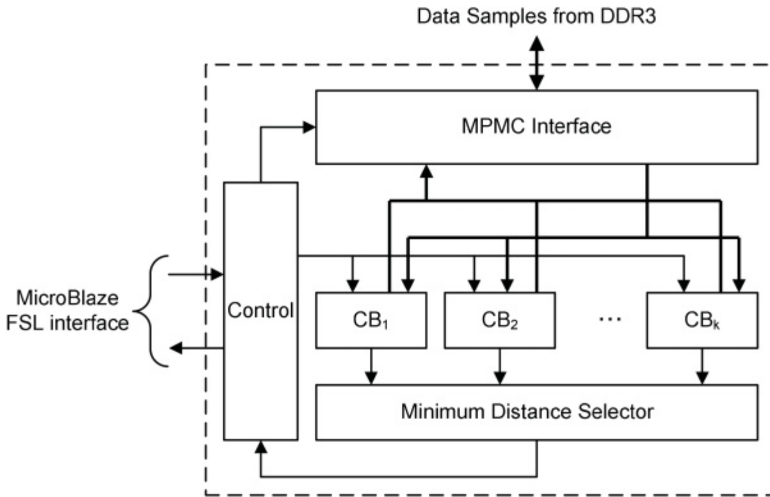

Fig. 3. Generic K-Means hardware accelerator architecture.

of each sample to the centroids of each cluster. The algorithm is used in a range of data-mining applications such as market segmentation analysis, computer vision, geostatistics, astronomy, and agriculture. It therefore represents a class of workloads that are increasingly being run on distributed, cloud computing platforms such as Hadoop. Although numerous FPGA implementations of the K-means algorithm have been proposed in the past [Lavenier 2000; Estlick 2011; Gokhale 2003; Saegusa 2006], we developed our own accelerators to ensure compatibility with the RASSD node's DDR3 MPMC and MicroBlaze FSL interfaces. We then compared the performance and energy efficiency of the RASSD node prototype customized with our K-means hardware accelerators to a multithreaded software implementation running on a commodity personal computer.

Figure 3 shows a generic K-means hardware accelerator architecture. Data samples are read from the external, DDR3 memory through the high-bandwidth MPMC interface circuit. Each data sample consists of a vector of m coordinates that is transferred to k centroid blocks (CBs). Each CB computes the distance between the data sample and its cluster centroid coordinates. Each CB also stores the centroid coordinates of its corresponding cluster. Once centroid distances are computed, they are processed by a minimum-distance selector block, which selects the CB corresponding to the smallest distance. This information is sent to a controller, which increments the cluster counter and accumulates the data sample coordinates in the appropriate CB. Once all data samples have been processed, the cluster centroid coordinates are recomputed. This is repeated until the cluster centroid coordinates stabilize.

4.2. Experimental Methodology

To evaluate the RASSD node architecture, we partitioned a 320MB dataset containing 40 million random points in a two-dimensional, Euclidean, space into 2, 4, 8, 16, and 32 clusters. We ran each experiment on two computational platforms. The first was a commodity personal computer (PC) with a 3.4GHz quad-core Intel Core i7–2600 processor and 8GB of RAM running under a 64-bit version of the Microsoft Windows 7 operating system. We used this platform to run a multithreaded implementation of the K-means algorithm (M.Russel) using one, two, four, and eight threads. The second platform was the RASSD node prototype configured with five implementations of the

Table I. FPGA Resource Utilization for Different Instances of the RASSD Node Prototype

FPGA Resources	Total Available	Static Components	K-Means Hardware Accelerators				
			K = 2	K = 4	K = 8	K = 16	K = 32
LUTs	150,720	9,528 (7%)	5,804 (4%)	11,608 (8%)	23,216 (16%)	46,432 (31%)	92,864 (62%)
Slice FFs	301,440	10,611 (4%)	14,790 (5%)	29,580 (10%)	59,160 (20%)	118,320 (40%)	236,640 (79%)
DSP48E1	768	3 (1%)	26 (4%)	52 (7%)	104 (14%)	208(28%)	416 (54%)
Clock Frequency		150MHz	200MHz				

Table II. Data Transfer and Cluster Centroid Compute Times

Clusters, K	PC (sec.)					RASSD (sec.)	
	Data Transfer	Computation				Data Transfer	Computation
		T = 1	T = 2	T = 4	T = 8		
2	36	53	26	26	26	40	5
4		106	53	26	26		
8		182	93	46	24		
16		380	194	97	51		
32		647	331	165	85		

K-means hardware accelerator. Each accelerator used a number of computational blocks that matched the required number of clusters. All accelerators were fully pipelined and operated at 200MHz. Both platforms used a 60GB OCZ Vertex Plus R2 solid-state drive running on a 285MB/sec SATA II interface.

4.3. FPGA Resource Utilization

Table I shows the FPGA resources used by the RASSD node when configured with different K-means hardware accelerators. The third column shows the resources used by the static components of the RASSD node. These include the MicroBlaze processor, cache memory, peripheral controllers, memory interface, system buses, and PRR interface. The fourth through eighth columns show the resources used by different instances of the K-means hardware accelerator. Because hardware accelerators are implemented inside a partially reconfigurable region, the region should be large enough to contain the resources required by the largest accelerator.

The number and complexity of the accelerators that can be implemented in a single FPGA device are limited by the amount of available logic. The FPGA device we used was only capable of supporting a maximum of 32 centroid blocks. This limitation can be overcome by distributing accelerators and processing data across multiple devices, or reusing accelerator structures in the same device to process data iteratively. This results in similar cost/performance tradeoffs to commodity, multicore processors with limits on the maximum number of threads they can run in parallel.

4.4. Execution Time

Table II shows the data transfer and compute times for both platforms. Our results show the significant impact hardware accelerators have in reducing cluster centroid compute times. The constant compute time in the RASSD node is primarily due to exploiting spatial parallelism to operate up to 32 centroid blocks simultaneously. Our results also show a large variance in PC cluster centroid compute times, which depend on the number of active threads and the number of clusters. However, hardware limits on the maximum number of active threads (8 for the PC vs. 32 for the FPGA), coupled with the sequential semantics of the von Neumann computational model, enable the RASSD node to compute cluster centroids almost 5 to 130 times faster than the PC

Table III. Power Consumed While Transferring Data and Computing Cluster Centroids

Clusters, K	SSD (Watts)	PC (Watts)				RASSD (Watts)
		T = 1	T = 2	T = 4	T = 8	
2	4.7	67.5	67.9	68.5	74.6	3.7
4						3.9
8						4.2
16						4.8
32						5.9

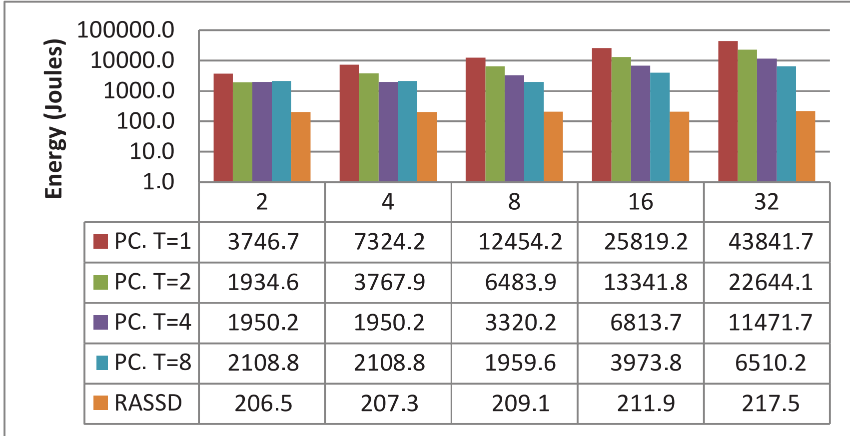

Fig. 4. PC and RASSD node energy consumption versus number of clusters.

despite running on a significantly slower clock. When data transfer times are taken into account, the RASSD node runs 1.3 to 15.2 times faster than the PC.

4.5. Power and Energy Consumption

We measured the average power consumed by the SSD and PC using a power meter. We also measured the average power consumed by our RASSD node prototypes directly from the ML605 FPGA (Texas Instruments, USB Interface Adapter Evaluation Module User's Guide 2006) (Texas Instruments, Fusion Digital Power Software). Table III shows our power measurements, while Figure 4 shows the total energy consumed by each platform as a function of the number of clusters. The results clearly show that the RASSD node consumes 9.4 to 201.9 times less energy than the PC due to its faster processing rates and lower power consumption. Figure 4 also shows that the PC wastes energy when the number of active threads exceeds the computational needs of the application. The ability to match the hardware accelerators of the RASSD node to application needs results in higher levels of energy efficiency.

In the remainder of this article, we develop an analytical model of an RASSD cluster platform to estimate its potential impact on performance when running Hadoop workloads. Because the performance of the platform depends on the characteristics of the hardware accelerators used in the RASSD cluster, and because designing and characterizing the performance of these hardware accelerators is very labor intensive, we use the results from this section as the basis for setting realistic hardware acceleration factors that we use in our analytical model. We present the details of our model in Section 5.

5. PROPOSED HADOOP MODIFICATIONS FOR RASSD MIDDLEWARE AND PROGRAMMING MODEL

In this section, we describe our suggested modifications that adapt Hadoop to the RASSD environment.

5.1. System Requirements

The overall objective for the Extended Hadoop system is to support the development and implementation of applications in the RASSD environment described in Section 3. From a user perspective, an application should be able to run in a mode similar to a standard Hadoop MapReduce environment. From a framework perspective, the system needs to support delegation of processing from the slave nodes to remote (RASSD) nodes capable of supporting kernel acceleration.

The execution for the application is initiated from the main() function, as shown in Table IV, in the master node just like in the MapReduce programming model. The main function calls the map, reduce, and combine functions and specifies the locations of the input and output directories. Once execution starts, the program is directed to the input file locations on the RASSD nodes, but these are not directly accessible. Instead they are accessed indirectly through slave middleware servers. Processing is directed to execute on the remote processing nodes, but these also, are not directly accessible from the standard Hadoop MapReduce framework. Instead, additional redirection is needed in the middleware slave nodes to direct the processing to the remote RASSD nodes. As a result, the Extended Hadoop system needs to support the following requirements:

1. A communication mechanism from the slave middleware servers to the RASSD nodes for sending processing requests, indicating which files to use, and receiving results.
2. Distributed File System Requirements additions to Hadoop's HDFS, which include:
   a. A metadata and index for the locations of the files on the remote processing nodes. This index will provide the ability to track the file locations on the remote RASSD nodes.
   b. A file management mechanism for the index files, which can track and update the index files as needed. When adding or deleting any file from any RASSD node, information about the file created/deleted has to propagate from the RASSD to the slave node and to the Master Middleware node.
3. MapReduce framework change requirements, which include:
   1) The processing of the map function has to run where the data is located, which is on the remote RASSD nodes. No data has to be moved from RASSD to MWS for map function processing.
   2) Ability to manage the collection and aggregation of results from RASSD nodes.
   3) A control mechanism for balancing processing requests on RASSD nodes since several jobs may run at the same time and several map tasks may require the same RASSD.
4. Balancing the processing load for improved overall system performance between the second level that contains the middleware servers and the third level of the system that contains the RASSD nodes.
5. Reliability: The system has to gracefully handle failures of RASSD nodes or failures of the tasks execution within the RASSD nodes. The reliability requirement is handled by a failure notification issued from the RASSD node or from the RASSD executor. The Task Tracker handles the failure of the task by rerunning the task using Hadoop original recover strategies.

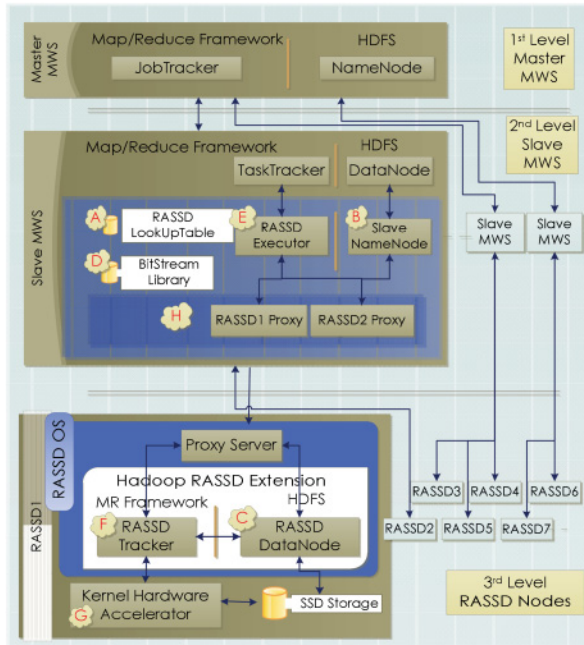

Fig. 5. Extended Hadoop architecture.

5.2. High-Level Architecture of the Proposed Extended Hadoop System

Here, we describe the high-level Extended Hadoop architecture. While the standard Hadoop has two layers (master and slave), the RASSD distributed environment requires a third layer of connection to the RASSD nodes. Figure 5 shows the proposed modifications for Hadoop to support the RASSD layer. The components labeled with alphabet letters are the new elements and are further described in this section. On the other hand, there are similarities to the standard Hadoop configuration. For example, the master node at the MWS layer contains the control components of the system such as the NameNode and the Job Tracker. In the new extended system, several components are added to the second layer of Hadoop that contains the slave nodes. The purpose of these components is to give the system the ability to communicate with the RASSD nodes to store data and execute tasks. The added components are in the form of classes to achieve certain needed functions and interact with Hadoop classes. Then new components are added to (1) HDFS, (2) MapReduce infrastructure, (3) Hadoop Common Libraries, and (4) Modifications for RASSD reliability. Here are the details:

1) HDFS Framework Modifications

We updated Hadoop's Datanode class to track the information about the files that are on the RASSD node. The updates enable the Data node to interact with the Namenode and manage the remote files as if they were local to the Slave node. In the second layer of slave nodes and the third layer of RASSD nodes, and as shown in Figure 5, the following additions are provided to extend the HDFS framework:

A. An RASSD Look-Up Table is placed in the second layer, and serves as an index for detailed information about the location of the files on the RASSD nodes. An additional field is included in this table to indicate the RASSDs that contain the needed files.

B. A Slave Namenode is added to the standard layer of Hadoop's slave nodes. The purpose of the addition is to manage files stored on the RASSD layer of the system. This node keeps a look-up table for the files' locations in the RASSD nodes.
C. An RASSD Datanode is added in the third layer to create equivalence to a Hadoop Datanode file management. In an RASSD node, all file system operations from creation to deletion are tracked through this RASSD Datanode. The collected metadata is communicated back to the Slave Middleware Server (SMWS).

2) MapReduce Framework Changes and Extensions
The TaskTracker class is updated to disable loading of the data at the Slave node level when running in RASSD mode. Additionally, code is included in the Task Tracker to request from the RASSD Executor a handler to the RASSD proxy. The programmer uses this RASSD handle to set the input parameters of the kernel. The following changes are also made to the second layer and the RASSD OS third layer to extend the MapReduce framework:

D. Accelerators' Library: On the slave nodes, a library of bitstreams is stored for sharing with the RASSD nodes. These libraries contain the bitstreams for special kernel kernels to be accelerated by the RASSD hardware. The application programmer can generate bitstreams separately, and then load these bitstreams to the HDFS.
E. RASSD Executor: To give Hadoop the ability to delegate the processing to the RASSD nodes, an "RASSD Executor" component is added to support sending a request with the appropriate kernel bitstream from the accelerators' library to the RASSD proxy to be configured on an RASSD node on demand.
F. RASSD Tracker: The purpose of this component is to create equivalence to a Hadoop Task Tracker in the external RASSD layer. The execution on the RASSD node is managed by this RASSD Tracker which manages the communication with the Slave MWS and the loading and execution of the hardware accelerator. It receives the map kernel (bitstream) through the proxy and loads it to the FPGA chip using dynamic partial configuration.
G. Kernel Hardware Accelerator: The kernel code is translated into hardware acceleration on the FPGA.

3) Hadoop Common Libraries Changes
Hadoop Common consists of the common utilities that support other Hadoop modules, like the MapReduce framework and the HDFS. We added the RASSD Proxies as a common utility to support the communication with the RASSD OS.

H. RASSD proxy: One proxy is added for each RASSD node connected to a middleware server. RASSD proxy is responsible for communicating with the RASSD node and reducing the communication complexity between the middleware servers and the RASSD nodes.

With embedded Linux OS running on our FPGA, the SMWS communicates with the node over a TCP/IP link. The OS manages the configuration of the hardware accelerators by loading their corresponding bitstreams, and provides basic file system operations to support slave data node functionality.

4) RASSD Reliability Changes
Hadoop has built-in mechanisms to deal with task failure by either rerunning the task on the same server or running the task on a different server with replicated data. To support RASSD reliability, the remote processing node needs to send a heartbeat to the slave node to show that it is alive and that it is running and healthy. The status information contains the percentage of data processed so far and the data left for

processing. When a task fails within the RASSD node, a message is sent to the slave node indicating that the task has failed and Hadoop uses its standard mechanism for recovery. When the whole RASSD node fails the slave node recovers all the data using the Hadoop standard recovery mechanism by rebuilding the data on another RASSD or Slave node using replicas in the HDFS, The tasks that were running on the failed RASSD node are recovered by having them reinitiated on another node.

5.3. Programming Model

This section describes the programming model as an extension of the standard MapReduce programming model. At the highest level, Extended Hadoop operates similar to the standard Hadoop. In the standard Hadoop, the map function contains the code for the tasks that need to be executed on the slave nodes. In the RASSD environment, in addition to the traditional map, reduce, and combine functions, the programming model is extended to include kernel functions. Ideally, we would want to have all the map functions accelerated by hardware, but it is not always feasible to transform all of the map function code into VHDL code for hardware acceleration. For example, operations to open a file inside the map kernel code should not be done by the hardware accelerator since the FPGAs processor runs at a fraction of the speed of modern processors. In the RASSD programming model, as shown in Figure 4, the map function is then divided into nonaccelerated code that runs on the slave middleware nodes, and the other part, called *kernel*, which runs on the RASSD node. The map function in the SMWS does not contain all the source code for the various tasks. It contains the drivelet code, which invokes one or more accelerated functions available on the reconfigurable fabric of the RASSD nodes. The FPGA CPU then executes the drivelets that initialize and launch the accelerators.

Our modified version supports backward compatibility to run code developed for original MapReduce and Hadoop without acceleration. The compatibility is enabled by invoking a switch that can disable sending data to the RASSD nodes. In this mode, the system reverts to processing on slave servers only.

5.4. Application Performance Model in RASSD Environment

In this subsection, we derive the application performance model in the Extended Hadoop environment, which is used in the evaluation section. Our Hadoop performance model is consistent with the detailed model in Herodotou [2011]. However, there are two differences: The first is that the new model has a modified modeling of map processing because it is processed on the third level of the system containing the RASSD nodes. The second difference is that our model presents aggregate measures of latencies and treats MapReduce as functional blocks of map, reduce, combine, or shuffle/sort. In regards to other internals of the Hadoop system that were not modified and may account for some latencies, the model does not account for these latencies of the components within Hadoop since they are not expected to be different. In what follows, the high-level model is described for job performance, followed by the performance model for the Map.

a) Job Performance Model

The total application execution time, T, is given by:

$$T = \sum_{i} (T_{job}), \tag{1}$$

Where i is the number of the MapReduce jobs in a chain-job application, where a chain-job application is a map reduce application that includes a sequence of map

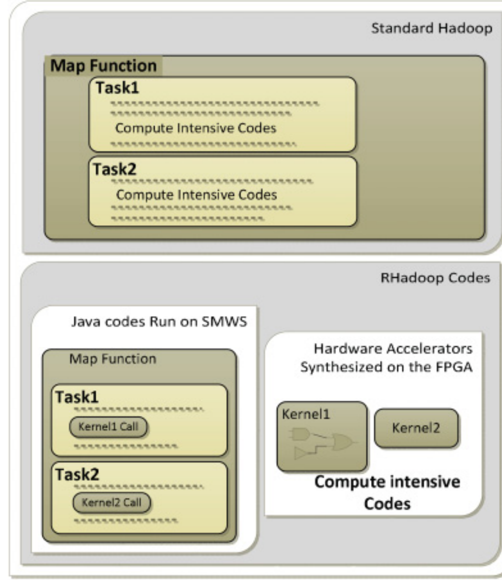

Fig. 6. Map function partitioning in the RASSD environment.

reduce jobs. For each job, the total time of the MapReduce processing is given by:

$$T_{job} = T_{Mf} + T_{SH} + T_{SO} + T_R + T_{ST}, \quad (2)$$

where T_{Mf} is the processing time for the map function, including the communication time of sending the results of accelerated processing from the RASSD node to the SMWS slave node; T_{SH} is the shuffling (communication between Slave Servers); T_{SO} is the sorting time; T_R is the processing time for the reduce phase; and T_{ST} is the scheduling time to load/store tasks, switch between tasks, and to check the look-up tables (LUTs) that guide the locations where the tasks need to execute. When comparing to an original Hadoop system, the total job processing time, T_{jobO}, is given by:

$$T_{jobO} = T_{MfO} + T_{SH} + T_{SO} + T_R + T_{ST}, \quad (3)$$

where T_{MfO} is the processing time for the map function in a standard Hadoop system. From Equations (2) and (3), the overall speedup for each job with the new system compared to a standard Hadoop is equal to

$$S_{job} = \frac{T_{jobO}}{T_{job}} \quad (4)$$

b) Map Performance Model

To further understand the overall performance gain, and because the map function is the main difference in job performance of extended system versus standard Hadoop, we further analyze the difference in map performance compared to a standard Hadoop. The map function in the new programming model is divided into two parts: nonaccelerated code, which runs at the Slave Middleware Server, and kernel code, which is accelerated and runs on the remote processing node (RPN) RASSD node.

The map function at the slave node calls the kernel through the proxy and relies on the RASSD proxy for communication with the RASSD node. Following Amdahl's law,

Table IV. Pseudocode of the Main Function in the New Extended Hadoop MapReduce Environment

Main Function	
1	Input: MasterDocument, Keyword
2	Set MasterDocument as dfsInputDirectory
3	Int JobNumber = 1;
4	Do JobConfiguration(Map-ReduceJob Job1)
5	Job1.setInput(dfsInputDirectory)
6	Job1.setOutput(dfsoutputDirectory)
7	Run Job1
8	JobNumber++;
9	dfsInputDirectory = dfsoutputDirectory
10	dfsoutputDirectory = SM‖JobNumber
11	While certain level not met
12	Output: List MatchedDocuments

the execution time of the map phase in the system can be expressed by the following equation:

$$T_{Mf} = (1 - p) * T_{MOriginal} + \frac{p}{N * H} * T_{MOriginal} + T_C, \tag{5}$$

where p is the portion that can be accelerated; (1−p) is the nonaccelerated portion of the map function; N is the number of RASSD nodes running the hardware accelerators; H is the hardware acceleration factor for the RASSD FPGA chip; $T_{MOriginal}$ is the execution time of the original map as a standalone; T_C is the communication time between the RASSD node and the Slave Hadoop Node. The execution time of the map phase in the system can be equally expressed as follows:

$$T_{Mf} = T_{MnonAcc} + \mathrm{Max}(T_{MKernelN-j}) + T_C, \tag{6}$$

where $T_{MnonAcc}$ is the execution time of the nonaccelerated portion of the map function and $T_{MKernelN\text{-}j}$ is the execution time of a kernel in the map function running in parallel with other kernel kernels on a RPN.

5.5. Example: String Matching

In this section, we use an application, called *String Matching* (SM) to illustrate the implementation in the Extended Hadooop environment. This example shows the ease of programming using our programming model. The implementation of SM requires multiple random accesses to the data. The application searches a master document for a keyword and returns a list of references if the document contains it. The process is repeated iteratively for all references. SM takes as input a title of a paper, called *master paper*, and a keyword of interest. The application then retrieves the document and then checks if the document contains the keyword. If the keyword is found, SM then searches in all the references that also contain the keyword. The successful searches are stored in a list, which we call the *matched list* to avoid repeated searches of document previously searched. In Abbani [2011], the experiments demonstrated, as was expected, that processing data on the remote nodes was faster than the cost of I/O data retrieving and processing. Here, we also assume that data resides on the remote nodes, and we illustrate the use of the Extended Hadoop environment and demonstrate that hardware acceleration makes Extended Hadoop superior to Hadoop.

The new implementation requires kernel kernels in addition to map, Combine, and Reduce functions. The Main function in Table IV shows the implementation of the driver function for Extended Hadoop. The pseudocode of the Main function looks just

Table V. Pseudocode for the MapReduce Job Configuration Function with the Document Search Application

JobConfiguration Function	
1	Input: Map/Reduce Job as job1
2	job1.setMapperClass(DocSearchMapper.class);
3	job1.setReducerClass(DocSearchReducer.class);
4	job1.setCombinerClass(DocSearchCombiner.class);
5	job1.setMapKernel(DocSearchMapKernel);
6	job1.ExecutionStyle(RASSD);
7	Output: configured job1

Table VI. Map Function Pseudocode of the Document Search Application

Map Function	
1	Input: Key/Value <PaperName K1, RefPName V1> ,Proxy_handler(P)
2	p1.selectKernel("DocSearch");
3	p1.SetParam("papername",V1);
4	p1.SetParam("keyword","SearchWord");
5	Vector References = p1.Run();
6	Foreach(Reference in References)
7	Output: key/value<V1, Reference>

like standard Hadoop's main function. The program starts by setting the configuration (line 4, Table I) of the MapReduce Job, which is explained later in Table V. This statement (line 4) is the key difference in the program in comparison to standard Hadoop. The call establishes the configuration of the MapReduce jobs on the Master Middleware Server to take as input the master document and the keyword, and to set the output as the list of identified matching references. The input and output Directories of the MapReduce job are then established, followed by the job execution. When the MapReduce job is finished, the Main Function copies the output directory into the input directory of the next MapReduce job and updates the name of the output directory.

Table V shows the configuration function for the MapReduce Job. The first three APIs (lines 2–4) are Standard Hadoop functions to select the map, reduce, and combine functions for the job. Lines 5 and 6 show the added special configurations for Extended Hadoop. Line 5 selects the kernel. Line 6, selects the execution style which can be either "Standard" for backward compatibility with Hadoop, or "RASSD" to run Extended Hadoop as we described in the previous sections.

The map function is shown in Table VI, and it runs on the SMWSs associated with the RASSD nodes containing the target data files. In the Extended Hadoop environment, the map function becomes a small function that contains calls to the application kernel kernels. Each input document is handled by at least one kernel based on the document file size. In the map function, the code lines 2 to 5 are specific to Extended Hadoop. These APIs are added to communicate with the RASSD Executor and the RASSD proxies. An RASSD proxy is instantiated by Extended Hadoop to intercept communication with the target RASSD node. Line 2 selects the kernel. Lines 3 and 4 set the parameters for the kernel, and line 5 delegates the execution of kernel function to the RASSD node. In this example, the RASSD executor uses an FTP connection to send the kernel to the RASSD, where the execution is managed by the RASSD Tracker.

It is important to note that several map tasks belonging to distinct search services may run on the same SMWS. Each proxy is able to handle several requests. The corresponding RASSD node runs multiple hardware kernels.

To support execution in the RASSD FPGA environment kernel code is written in a high-level language and then translated to VHDL and synthesized using FPGA design

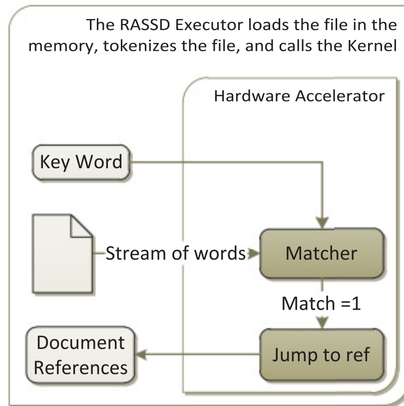

Fig. 7. Hardware kernel function synthesized on the FPGA chip.

Table VII. Reduce Function of the Document Search Application

Reduce Function	
1	Input: Key/Value <DocumentName K1, List of RefPName V1>
2	Output: key/value <k1, V1>

tools. The design of the hardware accelerator is shown in Figure 7, and it takes two inputs: the file name and the keyword to search for. A single output is expected, and it contains the references of the document.

For this particular application, there is no aggregation needed. However, there is a need to sort the data output from the different map functions. As a result, we just need to have the shuffle and sort and a dummy reduce function to ensure that step. The reduce function would just pass through the data already sorted, as shown in Table VII. The syntax of the reduce function is not modified from the standard MapReduce model.

6. EXPERIMENTS AND RESULTS

To verify our design, we conducted several experiments to demonstrate its feasibility:

1. The first set of experiments was conducted over three data sizes (small, medium and large) by using a simulated hardware acceleration to accelerate the map function at an RASSD RPN.
2. The second set of experiments is like the first but without acceleration.
3. One experiment was done to study the effect of multiple hardware accelerator speedups for different hardware implementations.
4. Several experiments were done for extracting the communication pattern of different applications to study the benefit of multiple RPNs on application performance.

For the sake of gaining flexibility in running the experiments, we used a software simulator, which was developed to reflect different hardware implementations for the same function with different speed ups. We used standard benchmark applications from the Phoenix MapReduce framework [Yoo 2009]. They are Word Count, String Matching, Inverted Index, KMeans, Matrix Multiplication, and Linear Regression. These applications form a good set since they have been historically used for benchmarking [Basaran 2013; He 2008; Fang 2011; Hong 2010; Ji 2011; Stuart 2011] to show the acceleration of MapReduce applications using GPU hardware. The primary differences between these applications are in the communication and computation complexity. Most of the MapReduce applications are batch applications that contain a sequence

Table VIII. Applications Data Sizes

	SM	WC	MM	LR	KM	II
Large	1.5GB	698MB	100×100×100 (200KB)	153MB	327MB	250MB
Medium	1GB	384MB	1,000×1,000×1,000 (22MB)	102MB	218MB	125MB
Small	0.5GB	128MB	1,500×1,000×1,500 (36MB)	51MB	109MB	57MB

of MapReduce jobs, so to address the effect of batch processing, we chose some of the applications implementations to have single a MapReduce job and others to have a batch of MapReduce jobs. All the applications were coded in Java.

6.1. Description of the Benchmark

The Phoenix benchmark [Yoo 2009] offers a variety of workloads and has an emphasis on communication and computation. This variation helps in evaluating the strengths and weakness of our proposed system. There are six applications from the benchmark that were used in testing. *Word Count* (*WC*) goes through a set of documents and calculates the number of occurrences for every distinct word. *Matrix Multiplication* (*MM*) calculates the product of two input matrices. *String Match* (*SM*) searches for a given keyword in a document. The application is a chain of MapReduce jobs, where sets of MapReduce jobs are called sequentially. Each Map task takes a line and searches for the keyword. The implementation uses a chain of jobs to show the effect of the jobs in the application. *Inverted Index* (*II*) builds the inverted index of the words in a set of documents. *KMeans* (*KM*) groups a set of vectors into K clusters according to their distances in the space. *Linear Regression* (*LR*) computes a linear model of a set of data. The data sizes associated with the applications used for testing are shown in Table VIII.

6.2. Experiment Setup

We used three experimental setups. The first was a simple Hadoop cluster with an RPN to simulate a RASSD node. The second was to collect the communication parameters between RPN and Hadoop DataNode. The third used the NS2 simulator to study the effect of many RPNs on the applications' performance.

a. End-to-End Setup for Hadoop with RASSD

The testing cluster for end-to-end testing is shown in Figure 9, consisting of one middleware server playing a dual role of master node and slave node, and one RPN simulating an RASSD node. Hardware acceleration is simulated through software delays. Ideally, we would need a reference for performance without acceleration and then simulate other systems with acceleration where performances can be achieved. However, due to the difficulty in actual hardware implementation, we chose the reference to be the system with maximum acceleration. We then needed to simulate slower systems using less acceleration, down to no acceleration. For this, we assumed the available RPN to be the base maximally accelerated system, and slower accelerations were then achieved by introducing delays in code. Master/Slave node contained Hadoop version 1.0.4 with Linux Ubuntu 12.10 and the RPN contained Linux Ubuntu 12.10 and Java code to simulate the RASSD processing. Additional code simulated a server socket for communication and the kernel function for processing. All data resided on the remote processing nodes (RASSD). The middleware server storage capabilities were used for storing the intermediate and temporary data.

b. Setup for Collecting Communication Parameters

The purpose of this setup was to collect the communication parameters for the benchmark applications. The network packet sniffer software was WireShark [Orebaugh 2006] on Ubuntu 12.10. WireShark records all the packets sent through the network. The setup is shown in Figure 8.

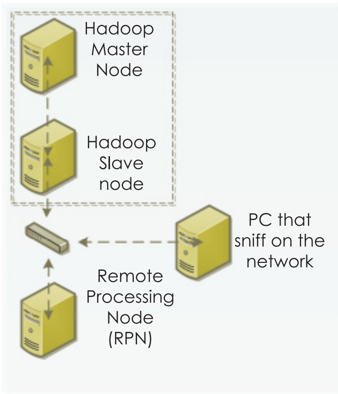

Fig. 8. Collect communication information to find the average communication bitrate.

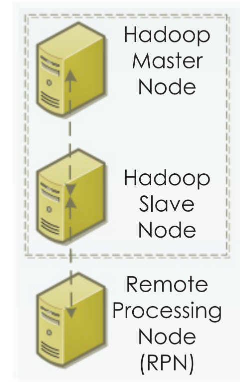

Fig. 9. Setup for end-to-end testing.

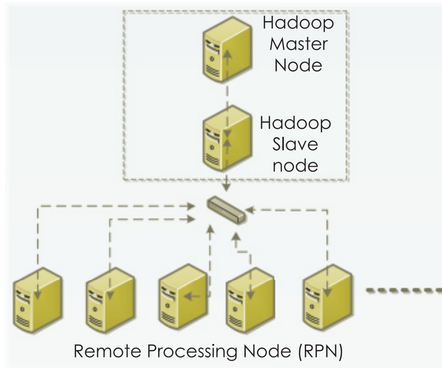

Fig. 10. NS2 simulation to find the optimal number of RASSDs for each application.

c. Setup to Simulate the Effect of Many RPNs with NS2 Network Simulator
The purpose of this setup was to simulate and assess the effect of a different number of RPNs on the applications' performance. We used the communication information collected using the previous setup to configure the NS2 Network Simulator [NS2 Network Siimulator 2008]. The NS2 configurations were as follows:

—The RPNs were set to be sources, since they already had the input data and they sent the results to the slave node.
—The slave Hadoop node was set to be the sink for all the connected RPNs.
—The bit rate between each RPN and the Slave node was set based on the communication pattern collected from the previous experiment setup.
—The number of nodes was increased in the experiment until the link between the switch and the Slave node started to drop packets, as shown in Figure 10.

6.3. Impact of RASSD Acceleration on Different Types of Applications and Loads

Two measurements sets were collected to evaluate the impact of acceleration on application performance as the data size increased. In the first set, raw latency and

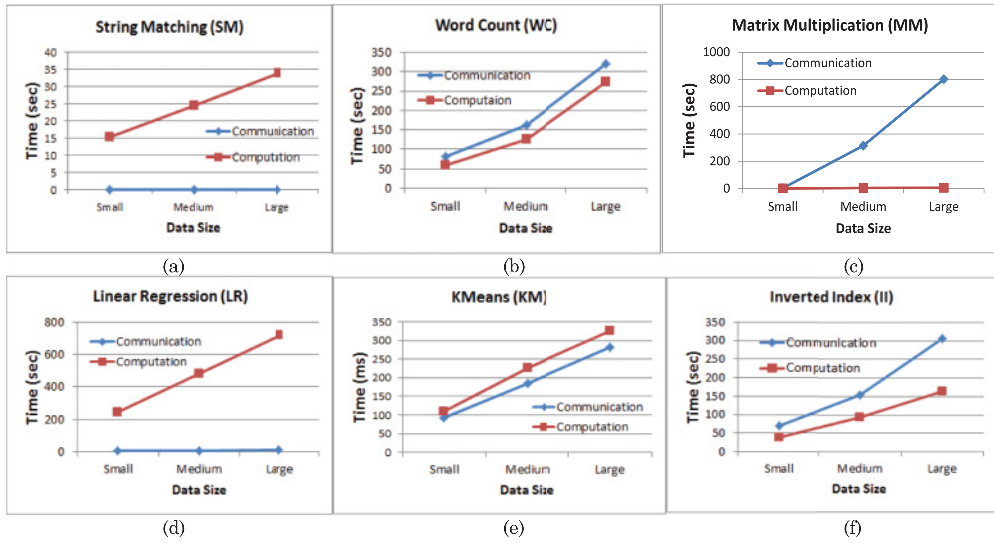

Fig. 11. Communication and computation trends in regards to increasing data size.

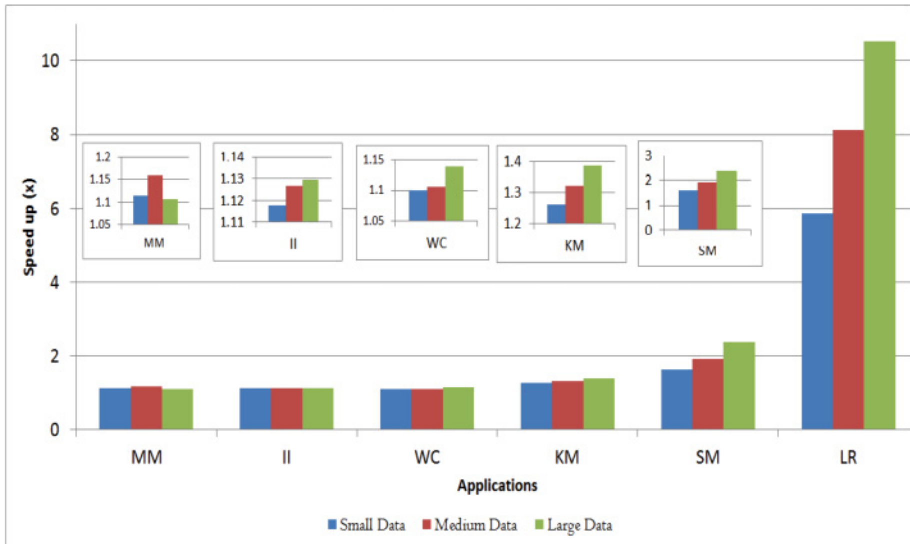

Fig. 12. Performance evaluation for acceleration effect in RASSD system.

communication measurements were collected, and the results are shown in Figure 11. In the second set, we measured the speedup gain with the system in comparison with a standard Hadoop, where we assumed a 20× hardware acceleration factor within the RASSD node. The results are shown in Figure 12. This factor is based on the accelerations we obtained for K-means, and this matches the results reported in the literature. For example, KMeans FPGA implementation achieved 51.7× acceleration in Hussain [2011], String Matching FPGA achieved 20× acceleration in Dandass [2008], and Matrix Multiplication FPGA achieved 18× acceleration in Dou [2005]. Section 6.5 shows the impact of varying acceleration factors.

The results in Figure 11 show different trends in computation and communication for different applications. It can be seen that for some applications, the computation cost increased with data sizes faster than the communication cost. For example, the LR application showed the least increase in communication. For some other applications, the communication is the bigger factor. As an example, the MM application showed the fastest rate of increase in communication and the slowest trend in computation. This trend can be attributed to more computations in the reducer as compared to the mapper. As a result and as shown in Figure 12, the LR application benefited the most from acceleration since it had the most complex mathematical operations in the map function and did not have much processing in the Reduce function. The computation factor of the LR application increased with the increase in input data size but the communication factor did not show a high rate with data size (Figure 11(d)). It can be seen that the speedup reached was $11\times$ for 153MB with the assumption of $20\times$ hardware acceleration. On the other hand, the MM application benefited the least from acceleration due to the extensive processing in the Reduce phase instead of the map phase. The role of the map function for the MM was reduced to only prepare the data for the multiplication implemented in the Reducer.

Figure 12 shows the measured application speed up gained by using hardware accelerators for applications with different data sizes. The testing showed the impact of data sizes on performance when we execute the map function on the RPN with and without acceleration. As seen in the figure, LR showed the highest gain for acceleration. The rest of the applications had a more balanced distribution of communication and computation factors with slight bias toward the computations. Some applications like KMeans and SM contained chained jobs. Performance was affected more by the scheduling factor T_{ST} (Equation (2)). This caused an increase in the total processing time and reduced the effect of acceleration in the map.

We conclude from the testing that different factors affect performance:

1. The bigger the *input data size*, the higher the speedup factor is. This effects the computation and communication trends of the applications, as shown in Figure 11.
2. The bigger the *map function code* is, the more time it consumes and the better performance we get when using hardware accelerators to accelerate this code.
3. The *communication* between each RPN and Hadoop slave node is a big factor that affects the speedup benefit that can be achieved with RPNs.
4. The more jobs in the application the bigger effect of the *scheduling factor* T_{ST} in Equation (2), and the lower application speedup we get from the acceleration.

6.4. Experiments for Characterizing Computation within the RASSD Environment

This section conducts a deeper analysis of the performance results and shows the breakdown for applications' processing time (map vs. reduce). The results are shown in Figure 14. Further breakdown of communication versus computation within the map function is shown in Figure 13.

As expected, LR showed a large fraction of the time spent in map processing, where most of the map processing time was attributed to kernel processing, as shown in Figure 14. This distribution supports the speedup observations in the previous section. On the other hand, the MM time breakdown shown in Figure 14 shows that most of the application time was spent for map processing, which also included communication with Hadoop's DataNode. Unfortunately, the time breakdown of MM's map function in Figure 13, showed that more than 95% of the map processing went into communication between the RPN and the slave node and less than 5% was the kernel processing time. These results further explain why the kernel acceleration of MM exhibited a small effect on the total application performance.

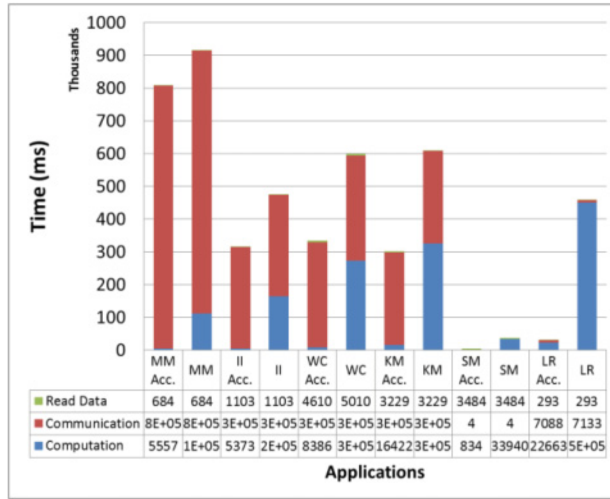

Fig. 13. Map functions' profiling.

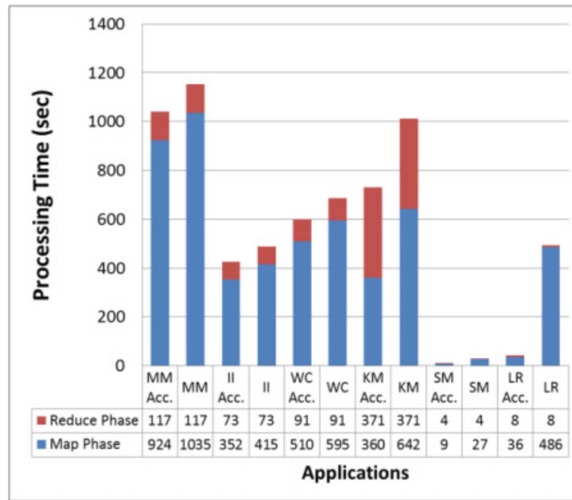

Fig. 14. Applications' profiling in terms of map and reduce.

6.5. Impact of Varying RASSD Acceleration on Overall Application Performance

To assess application speedup for different acceleration factors, we chose the LR application to help with highlighting the best performance that can be achieved with our benchmark applications since LR showed the best performance using kernel hardware accelerator. The data size was set to 100MB, and the acceleration speedup factor was varied from 1× (no hardware accelerator) to 20×. Figure 15 shows high application performance with faster accelerators. As expected from the performance model (Equation (6)), the main factors that affect the map function speedup are as follows: the data size D, the computational complexity of the map function (reflected by $T_{MKernelN}$), and the communication from RPN to the slave Hadoop node (reflected by T_C).

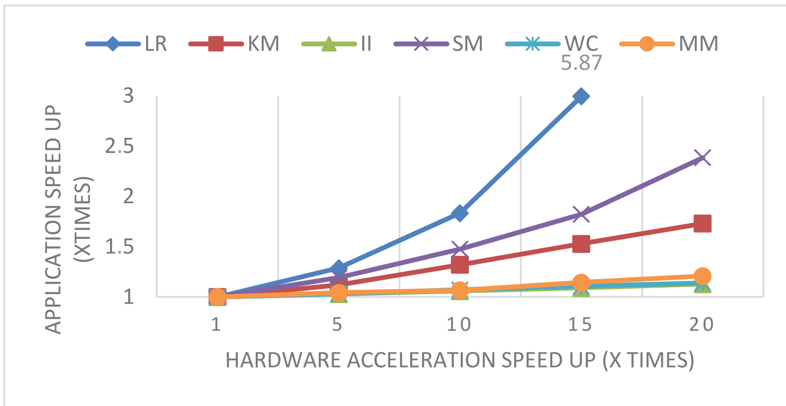

Fig. 15. Application performance with increasing the hardware accelerator speedup.

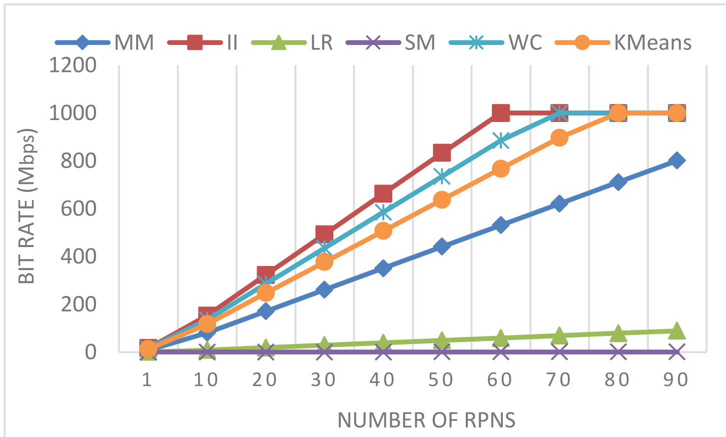

Fig. 16. Network occupation for application over different number of RPNs.

6.6. The Performance Effect of Different Number of RPNs

From Equation (5), the higher the number of RPNs is the higher the application performance that can be reached. The gain is achieved by distributing the tasks across the available RPNs assuming the data is already available on selected RPNs. Under ideal conditions, the number of RPNs, beyond which no additional benefit can be obtained, is proportional to the total amount of data that needs to be processed and inversely proportional to the maximum data that can be simultaneously processed on every RPN. For Hadoop, the default value for every RPN is 64MB for the block size. Then the number of RPNs would be the total size (in MB) of the input data divided by 64. There are additional factors that may reduce the need for more RPNs, like the network speed and communication pattern between the RPN and the Slave node; and the number of RPNs each slave node can control simultaneously.

7. CONCLUSION

In this article, we presented a new extension for Hadoop and the MapReduce programming model to support processing and cloud computing with a distributed hardware

acceleration environment. The RASSD environment was used as a case study. The map function contains calls to execute tasks on remote processing nodes where the data resides. The compute-intensive components, called *kernel kernels*, of the map functions are accelerated with dedicated hardware blocks on the remote processing nodes. The net effect is a major gain in performance due to hardware acceleration. In support of this extended MapReduce programming model, several components were added to the standard Hadoop. The design compatibility with Hadoop makes it easy to migrate MapReduce code from standard Hadoop to the new environment. Experiments with the benchmark applications showed the ease of implementations and potential for performance gains with the Extended Hadoop environment. The speedup for some applications reached 11× with 153MB with an assumption of 20× hardware acceleration. The application speed up is affected by the data size, the particular implementation, the complexity of the map function, and the communication trend of the map function.

Future work includes the development of utilities for evaluating the expected performance gain of new applications before implementing them in the new system. Additional future work may include the implementation of a hybrid platform that contains RASSD and GPU hardware acceleration.

REFERENCES

A. Abbani, A. Ali, D. Al Otoom, M. Jomaa, M. Sharafeddine, H. Artail, et al. 2011. A distributed reconfigurable active SSD platform for data intensive applications. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications (HPCC'11)*. 25–34.

A. F. Abbasi. 2012. A preliminary study of incorporating GPUs in the Hadoop framework. In *Proceedings of the IEEE 16th CSI International Symposium on Computer Architecture and Digital Systems (CADS'12)*. 178–185.

A. M. Acharya. 1998. Active disks: Programming model, algorithms and evaluation. *ACM SIGOPS Operating Systems Review* 32, 5, 81–91.

A. M. Ali. 2012. An operating system for reconfigurable active SSD processing node. In *Proceedings of the 2012 19th International Conference on Telecommunications (ICT)*. 1–6.

Amazon. (n.d.). GPU. Retrieved from http://aws.amazon.com/gpu/.

G. A. Attebury. 2009. Hadoop distributed file system for the Grid. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009.* IEEE.

C. A.-D. Basaran. 2013. Grex: An efficient MapReduce framework for graphics processing units. *Journal of Parallel and Distributed Computing* 73, 4, 522–533.

C.-T. S.-A. Chu. 2006. MapReduce for machine learning on multicore. *NIPS* 6.

Y. S. Dandass. 2008. Accelerating string set matching in FPGA hardware for bioinformatics research. *BMC Bioinformatics* 9, 1, 1–11.

J. A. Dean. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1, 107–113.

Y. S. Dou. 2005. 64-bit floating-point FPGA matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*. 86–95.

E. T. Elnikety. 2011. iHadoop: Asynchronous iterations for MapReduce. In *Proceedings of the 3rd International Conference on Cloud Computing Technology and Science (CloudCom'11)*. 81–90.

M. M. Estlick. (11, February). Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware. In *Proceedings of the 2001 ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays.* 103–110.

W. B. Fang. 2011. Mars: Accelerating MapReduce with graphics processors. *IEEE Transactions on Parallel and Distributed Systems* 22, 608–620.

G. A. Gibson. 1998. A cost-effective, high-bandwidth Storage Architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems.* 92–103.

M. J. Gokhale. 2003. Experience with a hybrid processor: K-means clustering. *Journal of Supercomputing* 26, 2, 131–148.

Hadoop. 2012. *Welcome to Apache Pig.* Retrieved from http://pig.apache.org/.

Hadoop. 2014. Welcome to Apache Hadoop. Retrieved from http://hadoop.apache.org.

HBase. 2014. Welcome to Apache HBase. Retrieved from http://hbase.apache.org.

B. W. He. 2008. Mars: A MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. 260–269.

S. C. Helmreich. 2006. *Data-centric Computing with the Netezza Architecture.* Technical Report SAND2006-3640, Sandia National Laboratories.

H. Herodotou. 2011. *Hadoop Performance Models.* Technical Report CS-2011-05, Duke Computer Science.

C. D. Hong. 2010. MapCG: writing parallel program portable between CPU and GPU. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques.* 217–226.

D. Hsiao. 1979. DataBase machines are coming, DataBase machines are coming. *IEEE Computer*.

H. M. Hussain. 2011. FPGA implementation of K-means algorithm for bioinformatics application: An accelerated approach to clustering Microarray data. In *Proceedings of the 2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS).* 248–255.

F. A. Ji. 2011. Using shared memory to accelerate MapReduce on graphics processing units. In *Proceedings fo the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*. 805–816.

M. K.-D. Jomaa. 2013. A mediation layer for connecting data-intensive applications to reconfigurable data nodes. In *Proceedings of the 2013 22nd International Conference on Computer Communications and Networks (ICCCN'13).* 1–9.

K. D. Keeton. 1998. A case for intelligent disks (IDISKs). *ACM SIGMOD Record* 27, 3, 42–52.

D. Lavenier 2000. *FPGA Implementation of the K-Means Clustering Algorithm for Hyperspectral Images.* Technical Report LA-UR # 00-3079, Los Alamos National Laboratory.

X. J. Liu. 2009. Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09).* 1–8.

Mahout. 2014. What Is Apage Mahout? Retrieved from http://mahout.apache.org/.

J. M. Manyika. 2011. *Big Data: The Next Frontier for Innovation*. Competition, and Productivity, McKinsey Global Institute.

A. A. Mendon. 2012. A high performance, open source SATA2 core. In *Proceedings of the 22nd IEEE International Conference on Field Programmable Logic and Applications (FPL).* 421–428.

NIMBIX Cloud. 2014. Homepage. Retrieved from http://www.nimbix.net/.

NS2 Network Siimulator. 2008. In T. I. Hossain, *Introduction to Network Simulator ns2*. Springer. Retrieved from http://www.isi.edu/nsnam/ns/.

A. G. Orebaugh. 2006. *Wireshark & Ethereal Network Protocol Analyzer Toolkit.* Syngress. Retrieved from http://www.wireshark.org/.

E. A. Ozkarahan. 1975. RAP—An associative processor for data base management. In *Proceedings of the National Computer Conference and Exposition.* 379–387.

PEER1. 2014. Homepage. Retrieved from http://www.peer1.com/.

E. Riedel. 1999. *Active Disks—Remote Execution for Network-Attached Storage*. Doctoral dissertation, Technical Report, CMU-CS-99-177, Carnegie Mellon University, Pittsburgh, PA.

E. C. Riedel. 2001. Active disks for large-scale data processing. *Computer* 34, 4, 68–74.

M. Russel. 2010. http://www.eee.bham.ac.uk/russellm/ee3j2/\\lab2\%202010/source/k-means.c.

T. A. Saegusa. 2006. An FPGA implementation of K-means clustering for xolor images based on KD-tree. In *Proceedings of the 2006 IEEE International Conference on Field Programmable Logic and Applications (FPL'06).* IEEE, 1–6.

T. C. Scofield. 2010. XtremeData dbX: An FPGA-based data warehouse appliance. *Computing in Science & Engineering* 12, 4, 66–73.

Y. B. Shan. 2010. FPMR: MapReduce framework on FPGA a case study of RankBoost acceleration. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* 93–102.

J. A. Stuart. 2011. Multi-GPU MapReduce on GPU clusters. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS).* 1068–1079.

S. Y. W. Su, L. H. Nguyen, A. Emam, G. J. Lipovski. 1979. The architectural features and implementation techniques of the multicell CASSM. *IEEE Transactions on Computers* 28, 6, 430–445.

Texas Instruments. 2006. *USB Interface Adapter Evaluation Module User's Guide.* Literature Number: SLLU093.

Texas Instruments. 2012. Fusion Digital Power Software. http://www.ti.com/tool/fusion_digital_power_designer.

J. H. Yeung. 2008. MapReduce as a programming model for custom computing machines. In *Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines (FCCM'08)*. 149–159.

R. M. Yoo. 2009. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. 198–207.

Received June 2013; revised January 2014; accepted February 2014