รายงานการวิจัย พัฒนานวัตกรรมและสิ่งประดิษฐ์

เรื่อง

# การออกแบบและพัฒนาต้นแบบเชิงพาณิชย์ มทส. ไอยรา คลัสเตอร์ รุ่น บี1
# (A design and development of the commercial prototype of SUT Aiyara Cluster - B1)

หัวหน้าโครงการ
ผู้ช่วยศาสตราจารย์ ดร. ชาญวิทย์ แก้วกสิ
สาขาวิชาวิศวกรรมคอมพิวเตอร์ สำนักวิชาวิศวกรรมศาสตร์
มหาวิทยาลัยเทคโนโลยีสุรนารี

ได้รับทุนอุดหนุนการวิจัย พัฒนานวัตกรรมและสิ่งประดิษฐ์จาก
กองทุนนวัตกรรมและสิ่งประดิษฐ์ สมเด็จพระเทพรัตนราชสุดาฯ สยามบรมราชกุมารี
ประจำปี พ.ศ. 2557

ผลงานวิจัยเป็นความรับผิดชอบของหัวหน้าโครงการวิจัยแต่เพียงผู้เดียว
กรกฎาคม 2565

## บทคัดย่อ

ไอยราคลัสเตอร์ รุ่น บี 1 เป็นคลัสเตอร์ประหยัดพลังงานโมเดลที่พัฒนาต่อจากไอยราคลัสเตอร์ Mk-I โดยทำการปรับฮาร์ดแวร์คอนฟิกกูเรชันในส่วนของคลัสเตอร์จริงและทำการปรับปรุงคลัสเตอร์เชิงตรรกสองระบบที่ทำงานอยู่บนฮาร์ดแวร์จริง

รายงานวิจัยฉบับนี้รายงานการพัฒนาและผลการทดสอบคุณลักษณะของไอยราคลัสเตอร์ B1 ในแง่มุมของอัตราการประมวลผลข้อมูลพบว่าหลังการปรับปรุงอัตราการประมวลผลข้อมูลของไอยราคลัสเตอร์ B1 สูงกว่า Mk-I ประมาณ 2 เท่า นอนจากนั้นรายงานฉบับนี้ยังมีผลการทดสอบแง่มุมของการใช้พลังงานของคลัสเตอร์ด้วย

## Abstract

The Aiyara B1 cluster is a model of the power-saving cluster that had been developed after the Aiyara Cluster Mk-I. The B1 cluster had been improved by re-designing the hardware configuration of the physical cluster and re-implementing the two logical clusters which run on top of the physical one.

This report contains the development and the experimental results of the characteristics of the Aiyara B1 cluster, mainly the data processing rate. After the redesign and improvement, it is found that the Aiyara B1 cluster is almost 2 times faster than the Mk-1 model. Additionally, this report also discusses the power consumption aspect of the Aiyara B1 cluster too.

# กิตติกรรมประกาศ

โครงการวิจัย พัฒนานวัตกรรมและสิ่งประดิษฐ์ เรื่อง การออกแบบและพัฒนาต้นแบบเชิงพาณิชย์ มทส. ไอยรา คลัสเตอร์ รุ่น บี1 ดำเนินการจนสำเร็จลุล่วงด้วยดี ด้วยความร่วมมือจากหลายฝ่าย โดยเฉพาะอย่างยิ่งกองทุนนวัตกรรมและสิ่งประดิษฐ์ สมเด็จพระเทพรัตนราชสุดาฯ สยามบรมราชกุมารี ที่ให้การสนับสนุนด้านเงินทุน ผู้วิจัยขอขอบคุณผู้เกี่ยวข้องทุกท่านมา ณ โอกาสนี้

# Table of Contents

# Chapter 1

# Introduction

Hadoop [1] has been a de facto standard for unstructured and semi-structured Big Data analytics recently. It contains two major components, namely the Hadoop Distributed File System (HDFS) [2] and the MapReduce engine. The technology behind HDFS is based on the concept of the Google File System [3], while Hadoop's MapReduce engine implements the Map-Reduce programming model [4]. Widely adapted to run both on the cloud and in corporate sections, Hadoop has been included in several software stacks for Big Data, such as CDH from Cloudera, MapR, and Hortonworks, for instance. Also in early 2014, Intel invested in the Big Data industry to further fund the development of Hadoop's ecosystem [5].

Unfortunately, the public cloud is not a safe place for storing corporate data. Many serve security exploits have been discovered and widely-spread ones, for example, the Heartbleed exploit [6], which may put data on the cloud in danger. Some of these security exploits are reported to be known by intelligence organizations years before being revealed to the public [7]. Thus, the cloud outside the corporate is not an option for processing sensitive data to date.

A corporate usually consider building its own data center to store its data as an alternative choice to the public cloud. However, a data center is an expensive investment, both in terms of constructing and operating. According to Hamilton's cost model of data centers [8], more than 35 % of the total cost is the building facilities, the electrical power, and the cooling operations. So, small or medium-size corporates may not be affordable to maintain their own ones.

Big Data clusters made with ARM-based system-on-chip boards would be a better choice compared to building a data center. Kaewkasi and Srisuruk reported that they successfully built a Hadoop cluster with this kind of board and used it to process Big Data in an acceptable time [9]. Besides their work, there were several efforts that also studied a cluster made with commodity ARM boards [10], [11], [12], [13].

However, an ARM cluster is known to suffer from low performance because of its power-efficient nature. In the previous work [9], there was a conclusion that the CPU power of the ARM cluster is the primary reason that prevents this kind of cluster from being used in the production environment. This performance limitation motivated the work described in this report to further optimize Hadoop and the related software stack on the new Aiyara B1 cluster. This work described in this report further exploring of an effort to reduce the CPU usage in other parts of the system to increase the overall performance of the cluster.

Although the whole software stack, including the operating system, and the Java Virtual Machine (JVM), was optimized, this work focused on the optimization of Hadoop, and Spark [14]. The research questions of this work are as follows. Firstly, how does the data integrity verification in HDFS affect the performance of the cluster? Secondly, how does the data compression used by Spark affect the performance of the cluster? Finally, how does the power consumption relate to this performance tuning?

To answer these questions, we propose a new hardware configuration of the Aiyara B1 cluster based on an ARM cluster reported in [9]. The performance improvement of the new model is described in this report. The filtered version of the most frequent word (MFW) program, also mentioned in [9], was used for benchmarking the cluster in this work. Also, the versions of Hadoop and Spark used in this work is the same one as bundled in Cloudera's CDH5 [15]. It was intended to demonstrate that the cluster could perform data analytics using one of the de facto Hadoop distributions.

The contributions of the work described in this report are as follows. Firstly, a set of performance improvement techniques, including their parameters, for Hadoop and Spark on the Aiyara B1 cluster, is presented. Secondly, the power consumption measurement of different software configurations is presented. It is also interesting to find that power consumption can reflect the behavior of the transformation stages of Spark. The remainings of this report are organized as follows.

Chapter 2 discusses related works. Chapter 3 presents the cluster machine, both in physical and logical forms. Chapter 4 discusses performance tuning techniques. Chapter 5 reports and discusses the experimental results, and this report ends with a conclusion and future works in Chapter 6.

# Chapter 2
# Related Works

A low-power cluster for Big Data has not been widely studied because of the reason of the inadequate power of this family of processors. However, there have been several works recently trying to develop and optimize this kind of cluster for Big Data.

## 2.1. Microwulf

Microwulf [20] has been presented in 2008 as a cluster designed to follow Beowulf's architecture. It was considered a cheap cluster because its price-to-performance ratio is under $100 per GFLOPS. The machine could be performing well under 25 degrees Celcius. Like the Beowulf cluster, Microwulf has been designed to be a computing cluster, so it is quite different from a Big Data cluster from the application's perspective. Microwulf's tasks are usually CPU-bound, while the tasks for a Hadoop cluster are often I/O bound.

## 2.2. Iridis-Pi

Iridis-Pi [10] is a small version, made of the Raspberry Pi Model B [21], of the Iridis super-computer at Southampton. Iridis-Pi consists of 64-node Raspberry Pi boards, each equipped with a 700 MHz processor. The Model B of Raspberry Pi has 512 MB of RAM. Similar to Microwulf, Iridis-Pi is a demonstrating cluster for intensive computation with MPI. Although a Raspberry Pi board has been successfully employed in these computing applications, it is not for an application like Big Data processing. Not only Raspberry Pi has a processor speed of 700 MHz but also has a limited amount of memory. A Hadoop cluster requires each node to run both HDFS and Spark software together, thus it is obviously too heavy for a Raspberry Pi board.

## 2.3. Whitehorn Cluster

This inefficiency of Raspberry Pi in the Big Data area has also been demonstrated on a 5-node cluster by Whitehorn [1]. The author reported that the size of Java's heap can be allocated very limitedly at 272 MB from the total memory of 512 MB. The author also mentioned that this kind of cluster could not be used in production. Later the developers at Cubietech [13], the company behind Cubieboard, formed an 8-node cluster using Cubiebard

A10, similar to ours. It was the first time that Hadoop, both HDFS and the MapReduce engine, could be used to perform trivial tasks but not for processing real-world data.

## 2.4. The Aiyara Cluster Mk-I

As the prior work of this work described in this report, Kaewkasi and Srisuruk reported that their 22-node Cubieboard A10 cluster has successfully processed a non-trivial size of data, 34 GB, in an acceptable time [9], where the median size of Big Data reported in [22] is approximately 15 GB. Instead of using Hadoop's MapReduce engine, they deployed Spark on top of the HDFS layer. They identified the inadequate CPU's power as the bottleneck of their Hadoop cluster and found that the CPU was spent mainly on the Hadoop's I/O.

Motivated by their works, the work described in this report further explores the characteristics of performance and power consumption of the new model, the Aiyara B1 cluster.

# Chapter 3
# Cluster Machine

This chapter describes the hardware and software configuration of the Aiyara B1 cluster. There are two logical clusters, an HDFS, and a Spark cluster laid on top of the physical cluster.

## 3.1. Physical Cluster

The physical cluster used in the work described in this report extends the best performance configuration reported in [9]. Each node in this new model, the Aiyara B1, is equipped with an ARM Cortext-A8 SoC board at 1 GHz with 1 GB RAM and 100 Mbps Ethernet PHY port. All worker nodes are connected with a 60 GB SSD each via the SATA port. From the previous work [9], the authors reported that a cluster with SSDs is less stable than one with mechanical drives. In this work, the cluster's stability had been observed and found that the less stability was caused by inadequate power supplies. When the cluster was starting up, its power consumption would be spiking to, at least, two times its idle state. This led to the random disconnection of SSD block devices, /dev/sda, on some nodes. A consequence would be the failure of some HDFS's DataNodes, and finally, the whole cluster could not continue working. After this finding, more power supplies were added to the cluster.

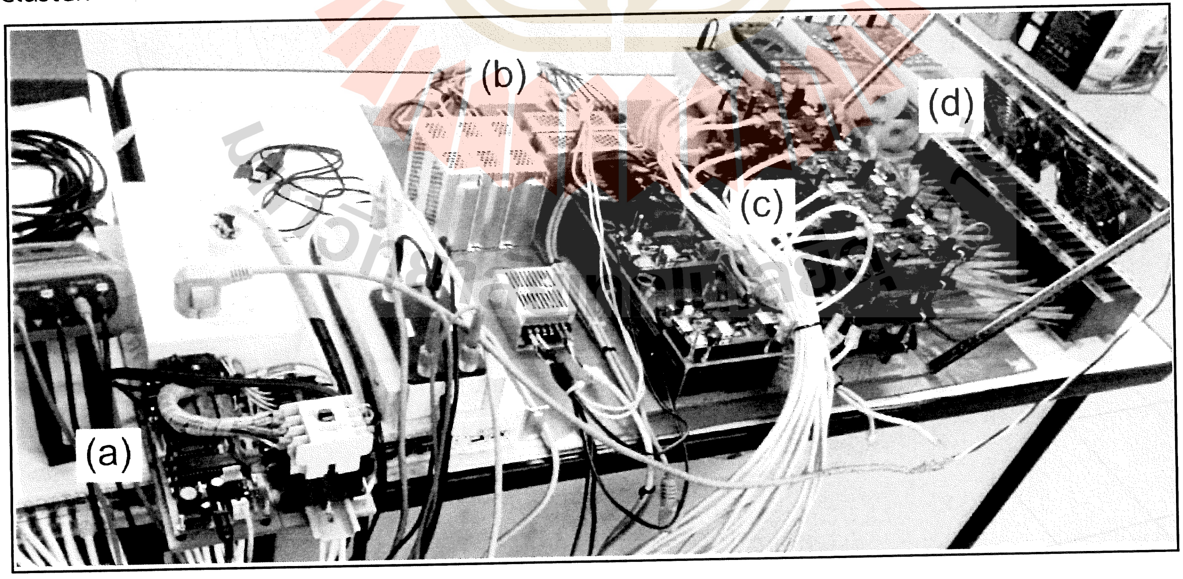

Fig. 1 - The hardware configuration of the Aiyara B1 cluster.

For the current hardware configuration, there were 5 power supplies for worker nodes. Each was divided to serve 4 nodes, i.e. 4 boards and 4 SSDs. The driver and the master node were powered separately by a small supply, as no power spiking was found from them.

Fig. 1 shows the hardware configuration of the cluster. Marker (a) is a Gigabit Ethernet switch connecting all nodes together. The 5 power supplies are located at marker (b). ARM SoC boards wiring together via their Ethernet ports are at marker (c). Boards on the left-hand side are the Spark Driver and the master node of both HDFS and Spark. On the right-hand side of marker (c) are the ARM boards. Each of them runs a Spark Worker and an HDFS DataNode. Each SSD attached to the ARM board is lined altogether in a row at marker (d).

## 3.2. Logical Clusters

The diagram in Fig. 2 illustrates the component architecture of the cluster.

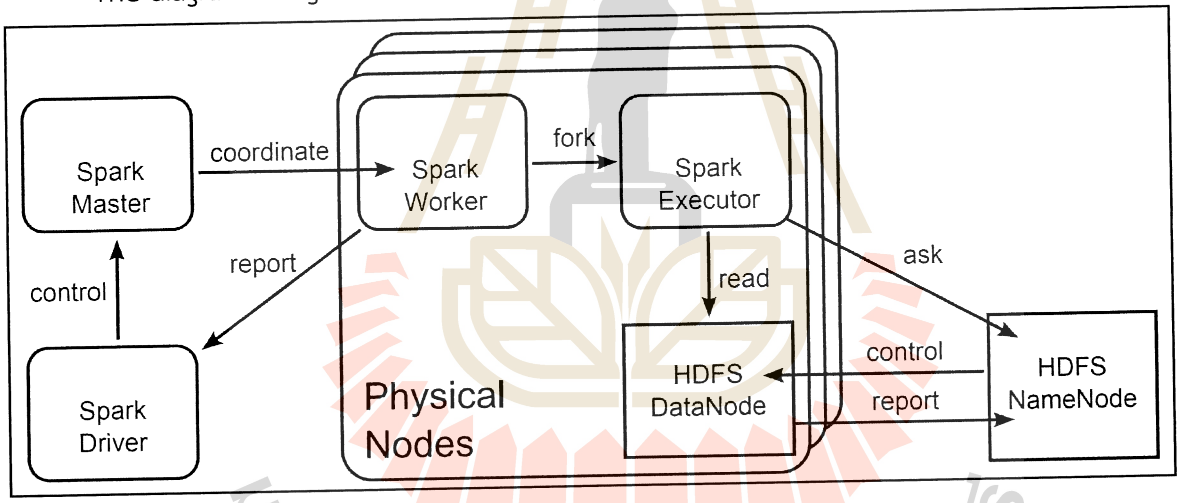

Fig. 2. The logical block diagram of the Aiyara B1 cluster.

There are both a Spark Worker and an HDFS's DataNode on the same physical node. A job execution will be starting from the Spark Driver node. It will submit a job to the Spark Master and the job will be serialized and sent to all available Spark Workers. Just before executing the job, the Spark Worker will fork a Spark Executor, which is responsible for processing the job's data. A Spark Worker is supervising one or more Spark Executors. If one of them dies, its supervisor will fork another Executor to recompute the lost tasks. To process the job's data, the Executor will ask about the data block's information from the HDFS's NameNode, located on the physical master node. Then, the Executor will read the actual job's data, 64 MB each, from the nearest's DataNode, which is usually on the same physical node. This locality level is called NODE_LOCAL in Spark. Shaded blocks in Fig. 2 represent the HDFS cluster, while others represent the Spark cluster.

## 3.3. Software Upgrades

Upgrading the software stack to Hadoop and Spark of CDH5 made the cluster unstable at first. There are several changes inside them that prevent the software to run out of the box on the cluster.

Firstly, the default block size in Hadoop version 2.3 is 128 MB instead of 64 MB as in version 0.20 [2]. This was the first cause that made the benchmarks fail. So, the block size value of 64 MB was selected in our experiments.

Secondly, the whole software package required a larger amount of memory compared to the previous work [9]. The first observation found was that a number of Spark Executors had died silently. It was later revealed that the Executor's processes were killed by the operating system as the whole system was running out of memory. This caused Spark Workers to try to re-spawn their Executors several times and eventually made the cluster fail to complete the job. This problem had been prevented by creating a swap partition on the built-in NAND flash device to allow a larger amount of memory.

Thirdly, we also found that the ext4 file system is not suitable to use as Spark's spill directory. A spill directory is used to temporarily store intermediate results during the data shuffling stages [16]. Spark will be creating a lot of directories and files, and sometimes performance degradation could be observed. In the worse case, a file or a directory was reported as not being found. With the swap partition prepared for Spark Executors, it was decided to move Spark's spill directory to a tmpfs file system which was mapped onto the virtual memory. With the tmpfs, Spark would then spill the intermediate results off-heap rather than directly to the disk. Then the paging out mechanism would help flush these results to the swap partition as big blocks, rather than a set of small files written to the ext4 file system. Fig. 3 describes the architecture before (a) and after (b) settings.

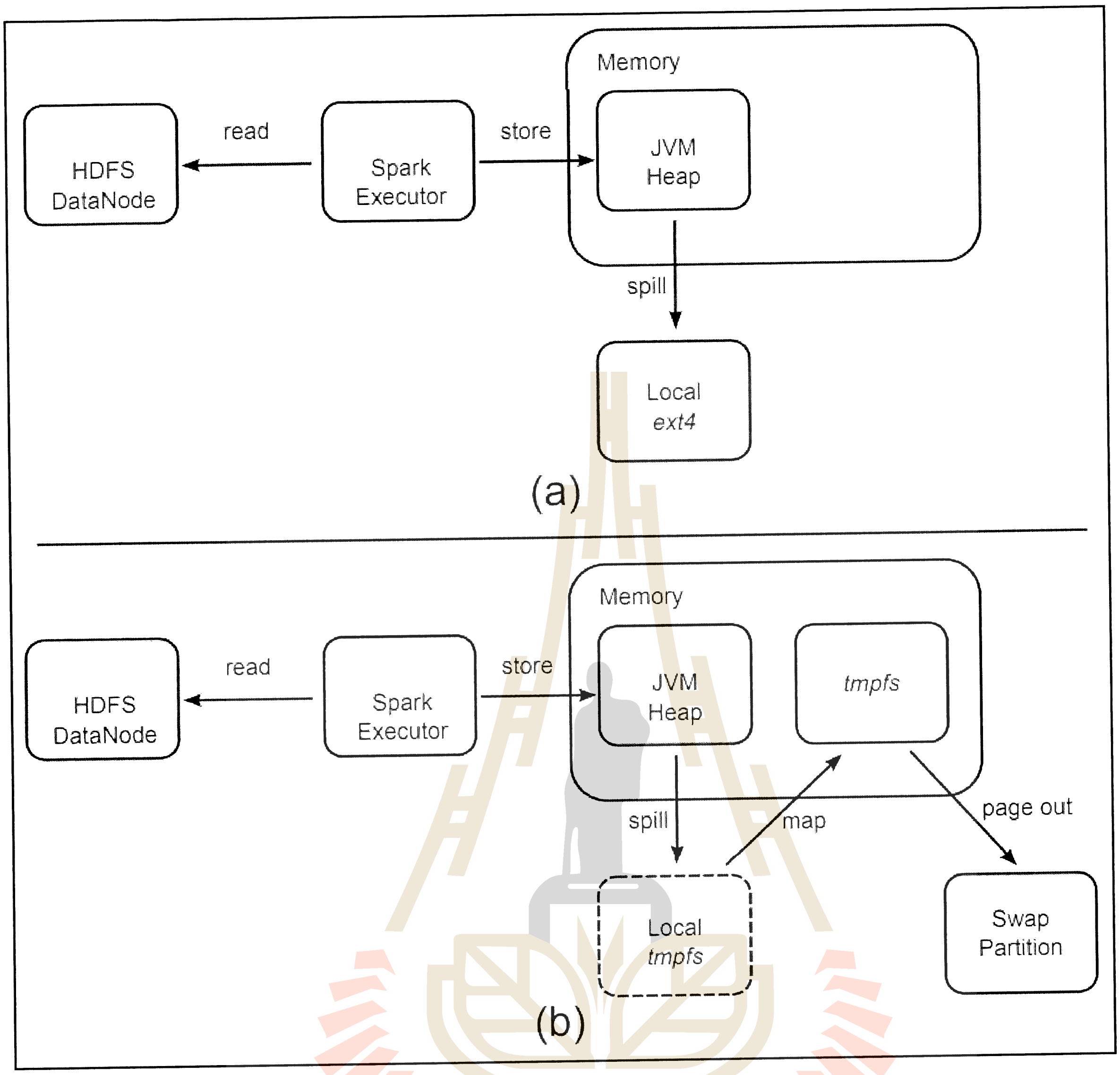

Fig. 3. The setting (a) before the performance improvement and the setting (b) after the improvement.

# Chapter 4
# Performance Tuning Techniques

This chapter discusses a number of performance tuning techniques applied to the cluster after it is made stable. The following topics, namely JVM, data integrity verification, and data compression, are tuned to boost the performance of the cluster.

## 4.1. Java Virtual Machine

Hadoop and Spark are written in JVM-based languages, so they run on the JVM. Practically, the parameters of the JVM affect the performance of Hadoop and Spark directly. Moreover, both Hadoop and Spark have an assumption that they will be running on server-class hardware, so their several default parameters are not fit for an ARM processor. To understand this, internal switches of the JVM are needed to observe.

The internal of Oracle's JVMs for ARM is studied and switches of the two virtual machines, the JDK 1.7.0 51 and the Embedded Java Runtime Environment (EJRE), were compared. All default switches of both JVMs were listed and the following was found. The JDK compiled for ARM contains both versions of HotSpot VM, the client (C1), and the server (C2), similarly to the desktop JDK. However, C1 is not compiled to be hard float. It does not support ARM's VFP nor ARM's NEON instructions. This unsurprisingly makes C1 very slow on an ARM board.

In contrast, C2 is compiled to be hard float, supporting both VFP and NEON. So the benchmarks were conducted on the cluster using C2, the server mode of the JVM. The EJRE is an optimization and commercial version built on top of the normal JDK. There are real-time-related switches in the EJRE and it disables UsePerfData by default. The EJRE contains only a single virtual machine. The JVM used in our experiments is the JDK and it is tuned using the same switches found in the EJRE.

4.2. Data verification with CRC32

Hadoop uses CRC32 to verify data integrity [2]. An HDFS's DataNode adds an extra 1-byte checksum for every 512-byte data block. Anyway, it is less than 1% extra storage required for integrity verification. In the early versions of Hadoop, there is only Java implementation of the CRC32 algorithms. Later in Hadoop 2.2, it is found that using a JNI library, libhadoop, to help compute CRC32 checksum will be faster, especially for a processor that supports special instructions for CRC32. Anyway, libhadoop does not support properly for the ARM architecture, especially on the armhf architecture. So, the build process of Hadoop is required a small modification to enable the libhadoop on ARM. A step further to try optimizing the CRC32 checksum is to patch libhadoop with NEON-enabled instructions [17]. So, in our experiments, there were 3 kinds of CRC32 computations, which are the Java implementation, the native libhadoop implementation, and the NEON libhadoop implementation.

Table 1 lists these 3 configurations in combination with data compression configurations, mentioned in the Data Compression section.

Table 1. All configurations that are used in the performance tuning techniques.

| Configuration Name | Description |
|---|---|
| java-nc | Disable libhadoop in both Hadoop and Spark, and disable compression in Spark. |
| java-cm | Disable libhadoop in both Hadoop and Spark, but enable compression in Spark with libsnappy. |
| native-nc | Enable the default libhadoop in Hadoop and Spark, but disable compression in Spark. |
| native-cm | Enable the default libhadoop in Hadoop and Spark, and enable compression, libsnappy, in Spark. |
| neon-nc | Use the NEON-based libhadoop in Hadoop and Spark, but disable compression in Spark. |
| neon-cm | Use the NEON-based libhadoop in Hadoop and Spark, and enable compression, libsnappy, in Spark. |

### 4.3. Data Compression

Data compression has been reported that improves performance in general cases, so it is enabled by default in Spark [16]. But there is an assumption that data compression requires extra CPU power, so it is not suitable for a constrained cluster formed with ARM processors. Data compression implementations that came with Spark are LZO and Snappy [16]. LZO uses Java's Unsafe package to improve performance, while Snappy uses JNI and required separated libsnappy. LZO failed to use because Java's Unsafe package does not work correctly on the ARM version of JVM, so we switched to Snappy. The default version of libsnappy has been built as a dynamic shared library, so it requires tweaking and rebuilding again. The libsnappy can be enabled by setting the compression codec of Spark to SnappyCodec. In our experiments, we used both compression and non-compression configurations in combination with other 3 CRC32 configurations. Thus, there were 6 different configurations for the experiments, as shown in Table 1.

# Chapter 5
# Experiments

This chapter describes the experiments and discusses their results. We will run performance evaluation experiments for all 6 configurations mentioned in Table 1 in chapter 4. We used the modified version of the MFW program, called cache-enabled MFW (CMFW), as the benchmark program.

## 5.1. Benchmark Program

Fig. 4 shows the CMFW program used in this work. As Spark's Executor is fault-tolerant by design [14], [19], its behavior of crashing, recovery and recomputation of lost tasks is a norm. This makes the whole cluster's behavior non-deterministic. It was found that we could not easily complete all benchmark stages without recomputation. To have a higher possibility of obtaining the exact experimental results, network-related parameters were set to be large enough for every node not to be timed out.

```
// sc is an object of SparkContext
var tf = sc.textFile("hdfs://...")
var r0 = tf.flatMap(_.split(" ").
                   matches("\\w+")).
    map(_, 1).
    reduceByKey(_ + _).
    cache()
var r1 = r0.map(_.swap).
    cache()
r1.sortByKey(false).take(10)
```

Fig. 4. The cache-enabled MFW program.

## 5.2. Performance

Fig. 5 shows the results of benchmarking 6 configurations of the cluster using the CMFW program. The results are directly collected from the Driver's console. The results are broken down as Spark's stages. According to Fig. 5, the largest portions of the results are in the first stage, reduce, as they are the parts the program spent for reading data from HDFS. Time spent for the sorting stages of all benchmarks is insignificantly different as they all involved shuffling a small amount of data, around 600-800 MB, over the network. All compression configurations, suffix denoted with -cm, used a larger amount of time compared to non-compression configurations, suffix denoted with -nc, because the former ones spent extra CPU time to compress the intermediate results at the reduce stage. The fastest configuration is java-nc, but processing times of all non-compression configurations are insignificantly different, anyway.

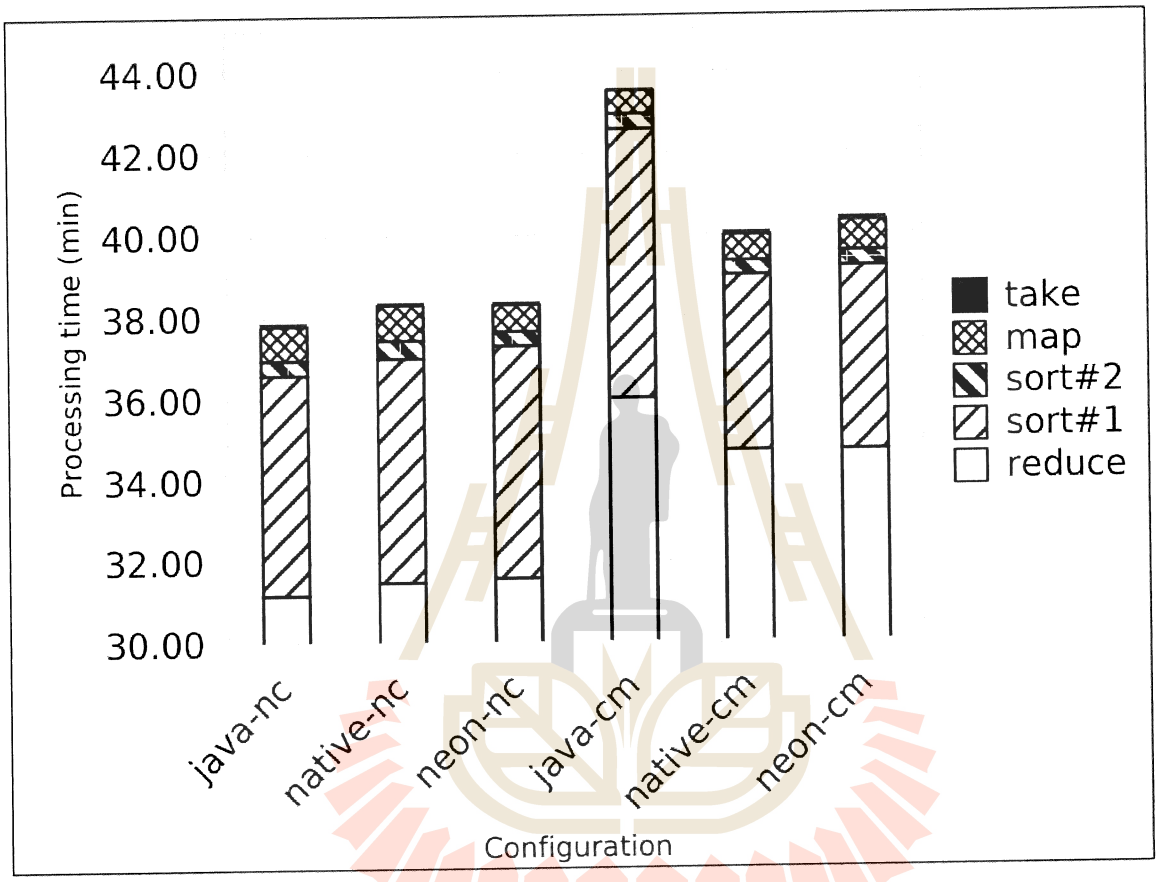

Fig 5. Processing time in minutes of all 6 configurations broken down by Spark's stages. Lower is better.

Fig. 6 shows the processing rate of all configurations in GB/min. The best configuration reported here is around 0.9 GB/min, almost 2 times faster than the results reported by the previous cluster, the Aiyara Cluster Mk-I.

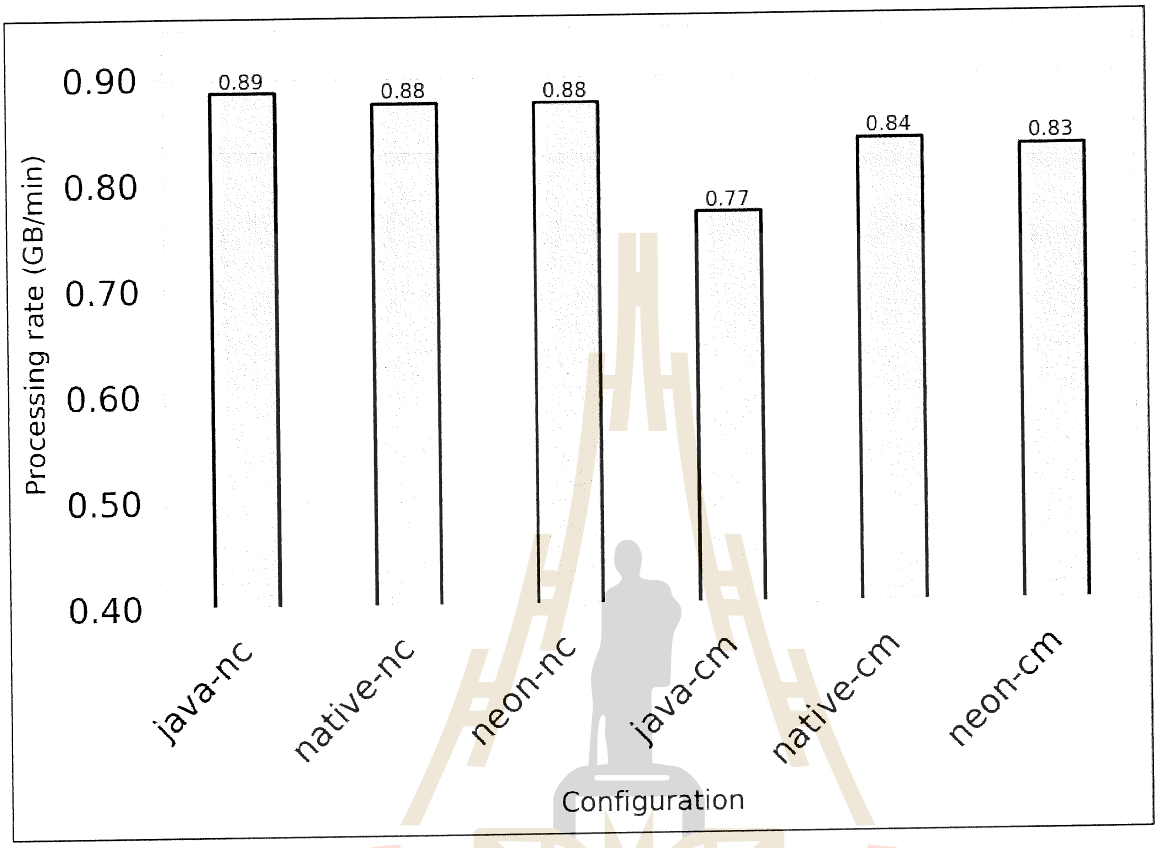

Fig. 6. Data processing rate in GB/minute of all 6 configurations. Higher is better.

5.3. Power Consumption

The energy consumption of each node is defined using the following equation,

$$e = V \cdot \sum_{n=1}^{k} I_n \Delta t,$$

where **e** is energy consumption in Wh, **V** is electric potential in Volt, $\mathbf{I_n}$ is electric current at index **n** in Ampere, and **k** is the time index of interest.

Fig. 7 shows averaged power consumption per physical node. The less power-consuming configurations are the NEON-enabled configurations. The main reason is that NEON is a specially designed instruction set for the ARM processor, so utilizing it makes the CPU consumes less power than Java configurations. The Java configuration with compression, java-cm, consumed the largest amount of power. This can be concluded for two reasons. First, the data compression requires extra CPU time, as previously mentioned. Second, the JVM requires code profiling and performs dynamic compilation during the execution, so the power is spent by the JVM on these activities. To better illustrate that the Java configurations consumed the larger amount of power, electric current consumption graphs are shown in Fig. 8.

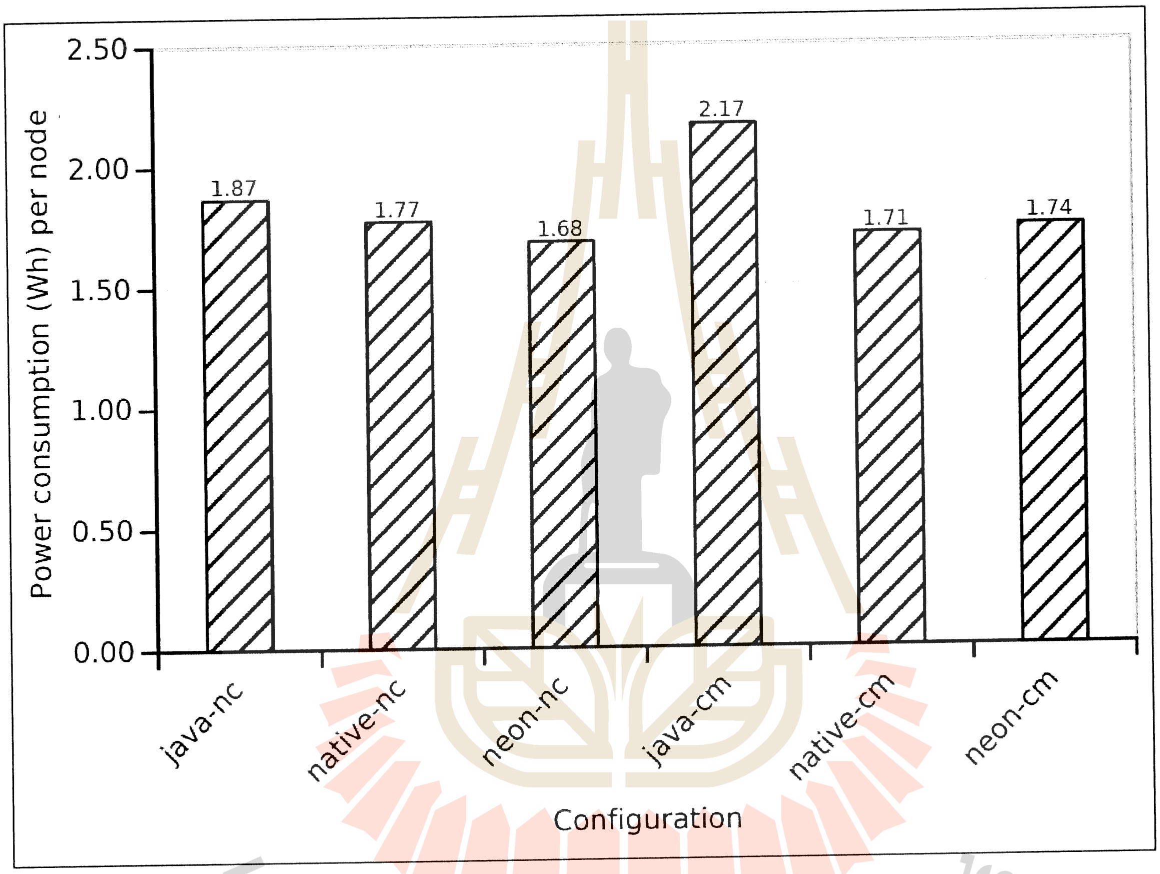

Fig. 7. Average power consumption per node for all 6 configurations. Lower is better.

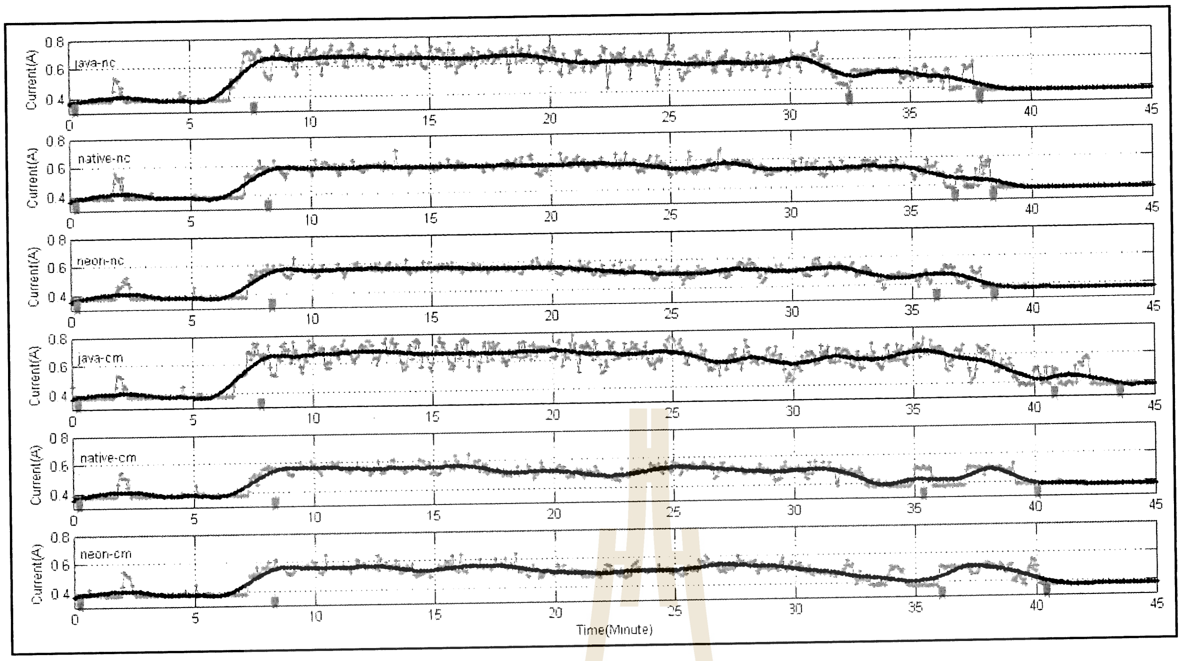

Fig. 8. The behavior of electric current consumption over time for all 6 configurations.

It is clear that the first and the fourth graphs for Java configurations, prefix denoted with java-, are less smoother than other configurations, especially in the first 30 minutes of processing. It is obvious from the internal behavior of the JVM. In contrast, the current consumption of the other 4 configurations is smoother as the HDFS layer does not use much power when libhadoop, which is written in the C language, is used by the cluster.

To summarize the power consumption of the cluster, Fig. 9 shows the peak and maximum average electric current consumed by each configuration. It is clear that NEON-enabled configurations, prefix denoted with neon-, are the best for energy saving. The processing rate and the power consumption ratio in Fig. 10 conclude that the best configuration balancing both performance and power consumption is neon-nc, the NEON-enabled configuration with non-compression.

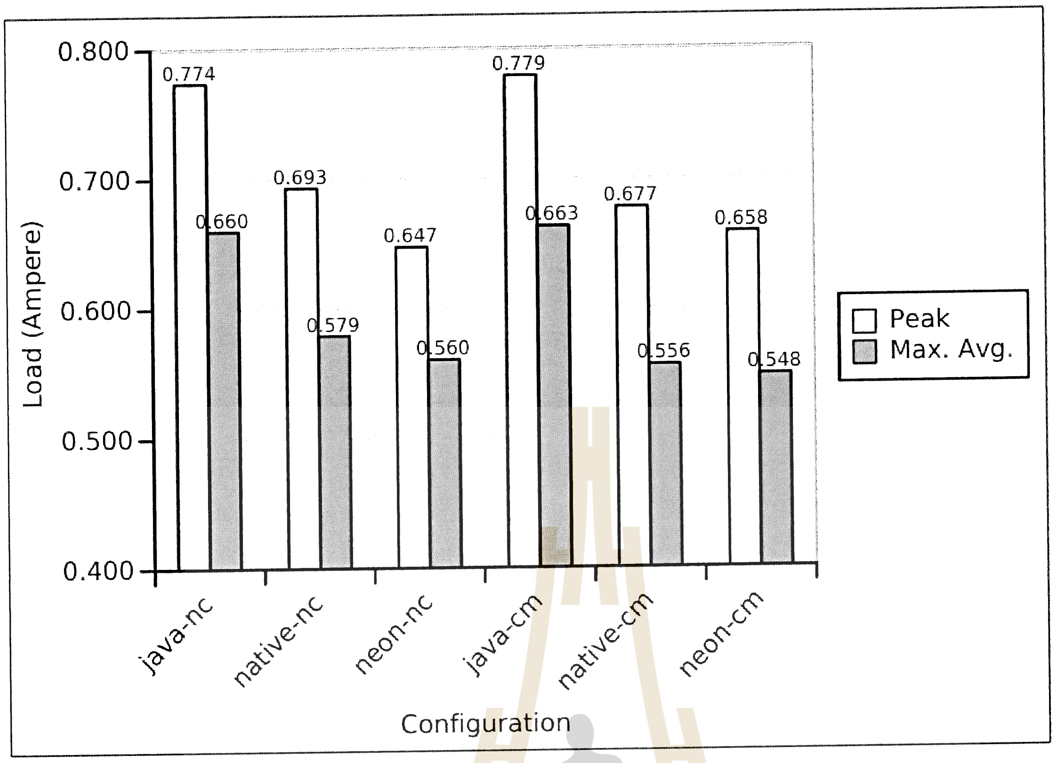

Fig. 9. Peak versus maximum average electric currents. Lower is better.

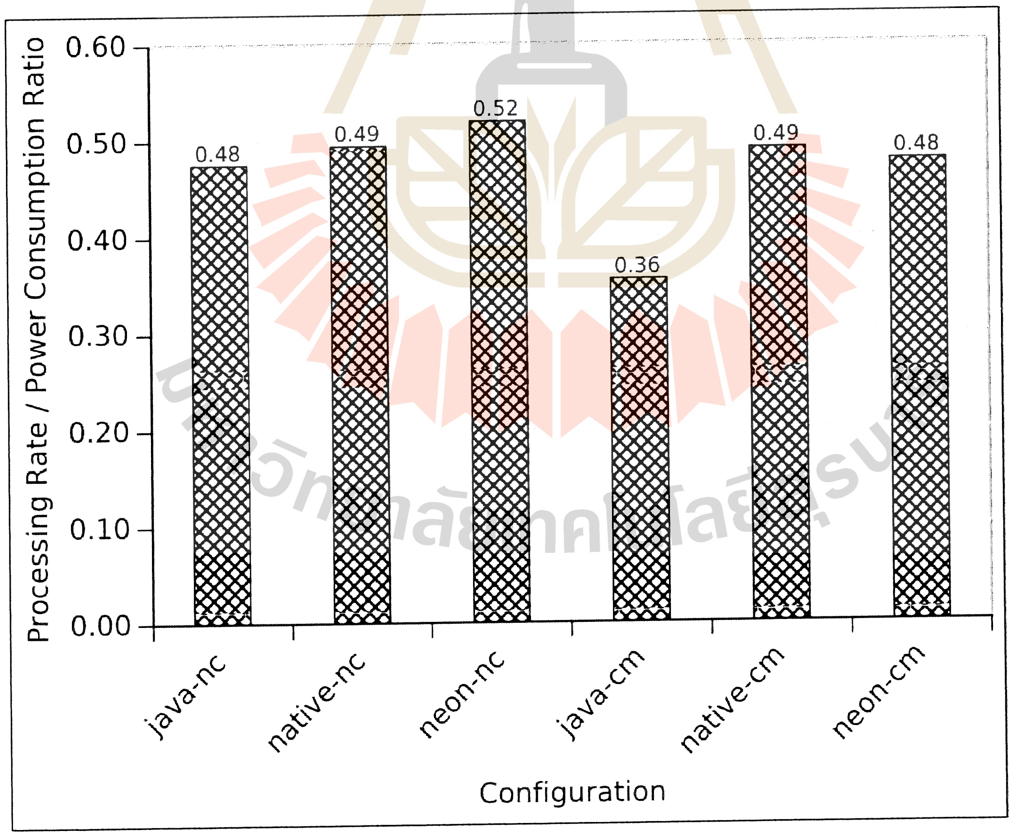

Fig. 10. Data processing rate per power consumption ratio. Higher is better.

# Chapter 6
# Conclusion and Future Works

This work described in this report presents a hardware configuration and a study of performance tuning of the Aiyara B1 cluster, an ARM cluster with SSD storage for data analytics using Hadoop's distributed file system and Spark. Then, the effect of performance tuning on power consumption is presented along with a set of tuning techniques. The techniques include re-architecting the software stack, tuning parameters of the JVM, and replacing parts of Hadoop and Spark with C libraries for both speeding data verification and data compression up. The results have shown that the fastest cluster configuration is almost 2 times faster than the previous results of the Aiyara Cluster Mk-I reported in [9].

The fastest configuration can process the same size of data, the Wikipedia article of size 34 GB, in roughly 38 minutes. Unsurprisingly, the configurations running only on the JVM consumed more power than those with C libraries. Data integrity verification with NEON instructions enables consumed less power than the native libhadoop. Data compression in Spark requires extra CPU time, so it is recommended to be disabled when processing data with ARM processors. The most balanced configuration is neon-nc, the configuration with NEON-enabled and non-compression.

New ARM system-on-chip boards equipped with massive co-processor cores, i.e. GPU, have emerged recently. With their low-watt and highly-parallel characteristics, it would be possible to explore an opportunity to process Big Data with the second-level parallelism provided by these massive co-processors.

# References

[1] T. White, Hadoop: The Definitive Guide, 1st ed. O'Reilly Media, Inc., 2009.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), May 2010, pp. 1–10.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29-43.

[4] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Commun. ACM, vol. 51, no. 1, pp. 107-113, Jan. 2008.

[5] N. Randewich. (2014, Mar.) Intel invested $740 million to buy 18 percent of Cloudera. [Online]. Available: http://www.reuters.com/article/2014/03/31/us-intel-cloudera

[6] [6] MITRE Corporation. (2014) CVE-2014-0160. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[7] R. Lawler. (2014, Apr.) Cloudflare challenge proves worst-case scenario for Heartbleed is actually possible. [Online]. Available: http://www.engadget.com/2014/04/11/heartbleedopenssl-cloudflare-challenge/

[8] J. Hamilton, "Cooperative expendable micro-slice servers (CEMS): low cost, low power servers for internet-scale services," in Conference on Innovative Data Systems Research (CIDR), 2009.

[9] C. Kaewkasi, and W. Srisuruk, "A Study of Big Data Processing Constraints on a Low-Power Hadoop Cluster," in 2014 IEEE International Computer Science and Engineering Conference (ICSEC). Khon Kaen, Thailand: IEEE Computer Society, 2014, pp. 308–313.

[10] S. J. Cox, J. T. Cox, R. P. Boardman, S. J. Johnston, M. Scott, and N. S. OBrien, "Iridis-pi: a low-cost, compact demonstration cluster," Cluster Computing, pp. 1–10. [Online]. Available: http://link.springer.com/article/10.1007/s10586-013-0282-7

[11] J. Whitehorn. (2013, Nov.) Raspberry flavored Hadoop. [Online]. Available: http://www.iDataSci.com/1/post/2013/11/stratu-euignite-slides-raspberry-flavoure d-hadoop.html

[12] Y. Wang. (2013, Dec.) Hadoop MapReduce performance study on ARM cluster. [Online]. Available: http://www.slideshare.net/airbots/ hadoop-mapreduce-performance-study-on-arm-cluster

[13] Cubietech Limited. (2013) Hadoop on Cubieboard. [Online]. Available: http://cubieboard.org/2013/08/01/hadoophigh-availabilitydistributed-object-orien ted-platform-on-cubieboard/

[14] M. Zaharia, et al., "Spark: Cluster computing with working sets," in Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, p. 10-10.

[15] Cloudera Inc.. (2014) CDH 5 Release Notes. [Online]. Available: http://www.cloudera.com/content/cloudera-content/clouderadocs/CDH5/latest/ CDH5-Release-Notes/CDH5-Release-Notes.html

[16] Apache Spark. (2014) Spark Configuration. [Online]. Available: http://spark.apache.org/docs/latest/configuration.html

[17] S. Capper. (2013, Mar.) Hadoop DFS performance. [Online]. Available: http://www.slideshare.net/linaroorg/lca13-hadoop-1

[18] M. Odersky, et al., "An overview of the Scala programming language," in Technical Report IC/2004/64. EPFL, Lausanne, Switzerland, 2004.

[19] M. Zaharia, et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12. Berkeley, CA, USA, 2012, p. 2-2.

[20] J. C. Adams and T. H. Brom, "Microwulf: A Beowulf cluster for every desk," in Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, ser. SIGCSE '08. New York, NY, USA: ACM, 2008, pp. 121-125.

[21] E. Upton and G. Halfacree, Raspberry Pi User Guide. John Wiley & Sons, 2013.

[22] A. Rowstron, et al., "Nobody ever got fired for using Hadoop on a cluster," in Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing, ser. HotCDP '12. New York, NY, USA: ACM, 2012, pp. 2:1-2:5.