GREENMAP: MAPREDUCE WITH ULTRA-HIGH-EFFICIENCY POWER DELIVERY

BY

DU SU

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Associate Professor Yi Lu
ABSTRACT

With the continuous growth of online services, energy consumption has become a significant fraction of the total cost of ownership of large data centers. Though much work in green computing has focused on improving efficiency for computation units such as CPU’s or servers, little attention has been paid to power delivery structures, such as voltage converters, which takes 10-20% of total energy consumption even before any computation takes place. Recently, a new power delivery architecture called *series stack* has been proposed in the power community, aiming to reduce conversion power loss. In series stack, servers are connected serially, and differential converters are used to regulate server voltage.

However, to effectively reduce conversion loss in series stack, computation loads need to be balanced in real time. To balance load for series stack, we implemented *GreenMap*, a modified MapReduce framework on top of series stacks, that assigns tasks in synchronization. We evaluated the conversion loss of GreenMap on a small data center. At all loads, GreenMap achieves a 81x-138x reduction in conversion loss from commercial-grade high voltage converters used by today’s data centers. The saved power is equivalent to 15% reduction in total energy consumption. GreenMap also achieves 67%-80% reduction in conversion loss compared to Hadoop’s FIFO scheduler under serial stack structure. Based on the observation that the average response time of GreenMap suffers a degradation at low load, we further propose a modification of GreenMap with dynamic scaling to achieve a favorable tradeoff between response time and power efficiency.
To my dear parents for their love and support.
ACKNOWLEDGMENTS

We thank Enver Candan for his assistance with setting up the Yokogawa
wt310 digital power meter. This report is advised by Prof. Yi Lu, supported
by NSF grant CNS-1150080.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER 2 BACKGROUND</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Conventional Stack</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Series Stack</td>
<td>4</td>
</tr>
<tr>
<td>2.3 MapReduce and Hadoop</td>
<td>6</td>
</tr>
<tr>
<td>CHAPTER 3 GREENMAP: LOAD BALANCING ALGORITHM</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Current Profiling</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Synchronized Assignment for Homogeneous Task</td>
<td>11</td>
</tr>
<tr>
<td>3.3 Synchronized Assignment for Heterogeneous Tasks</td>
<td>13</td>
</tr>
<tr>
<td>CHAPTER 4 GREENMAP: IMPLEMENTATION</td>
<td>16</td>
</tr>
<tr>
<td>4.1 General Structure: Master-Slave</td>
<td>16</td>
</tr>
<tr>
<td>4.2 Heartbeat Protocol</td>
<td>16</td>
</tr>
<tr>
<td>4.3 Scheduler Implementation</td>
<td>17</td>
</tr>
<tr>
<td>CHAPTER 5 EVALUATION</td>
<td>19</td>
</tr>
<tr>
<td>5.1 Experiment with Homogeneous Tasks</td>
<td>19</td>
</tr>
<tr>
<td>5.2 Experiment with Heterogeneous Tasks</td>
<td>23</td>
</tr>
<tr>
<td>5.3 GreenMap with Dynamic Scaling</td>
<td>25</td>
</tr>
<tr>
<td>CHAPTER 6 CONCLUSION</td>
<td>27</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>28</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

2.1 A typical of DC power distribution map in data centers . . . . . . . . 3
2.2 Series stack architecture for a four-server stack . . . . . . . . . . . 6
2.3 Server-to-virtual-bus topology . . . . . . . . . . . . . . . . . . . . . . . . 7
2.4 Data center with series-connected stacks . . . . . . . . . . . . . . . . . . 8
2.5 MapReduce job example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.6 Hadoop structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3.1 Current consumption of a word-count job with one map task
and one reduce task . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.2 Current profiles of four servers . . . . . . . . . . . . . . . . . . . . . . . . 15

5.1 Serial stack and measurement equipment used in the experiment . . 20
5.2 GreenMap reduces power conversion loss from the conventional
architecture by two orders of magnitude . . . . . . . . . . . . . . . . . . . . . . . . . . 21
5.3 GreenMap achieves response time comparable to that of Hadoop
FIFO scheduler at load 0.6 and above, while increasing response
time at lower loads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
5.4 The statistics of job size with Pareto distribution . . . . . . . . . . . . 23
5.5 Reduction in power loss of GreenMap and FIFO on serial stack . . . . 24
5.6 Response time of GreenMap and FIFO on serial stack . . . . . . . . 25
As our reliance on online services continues to grow, data centers hosting these services scale both in size and hardware grade. One main consequence is the increasing energy consumption in data centers. Nowadays, energy consumption has become a significant fraction of the total cost of ownership (TCO) in large data centers: by the year 2007, 2-3% of the global carbon emission was from data centers [1]; by the year 2009, electricity bills for data centers totaled nearly one million dollar per month [2]; in the year 2013, 91 billion kWh energy was consumed by data centers [3]. Thus, designing environmentally friendly data centers is an urgent need [4].

To design green data centers, much work has focused on improving computational energy efficiency, i.e. energy consumed by each unit of computation [5, 6]. For instance, through request redirection or virtual machine migration, one can consolidate demand onto a small portion of servers [7, 8, 9] and save energy by idling other machines; through speed gating for each server, one can optimize individual power usage [10].

However, all the improvements assume the structure of today’s data center, which adopts a conventional power delivery structure that is designed for single server applications. This structure requires a large step-down from the grid voltage of 600 or 480V AC [11] to a low voltage used by servers, typically 12V DC. With today’s data center technique, the step-down is realized by several power converters, and the conversion efficiency is limited to 80 – 90% [12, 11]. This means 10 – 20% of total energy consumption, i.e. the conversion loss, is wasted before any computation takes place.

Recently, a new power delivery architecture called series stack has been proposed in the power community [13], aiming to reduce conversion loss in data centers. In this architecture, servers are connected serially to stack power supply in order to avoid the large step-down, and a differential converter is used to regulate the voltage for each server. However, the conversion
loss in differential converters largely depends on the difference in computational loads across the servers. Thus, to reduce the conversion loss in series stack, the loads in the stack need to be balanced. It was demonstrated in [13] that with all servers running the Linux “stress” utility, hence an almost perfectly balanced load, 99.89% power efficiency is achieved. Yet no realistic data center traffic, which is much more unbalanced, has been studied.

In this thesis, we explore the feasibility of series stack for data centers with parallel data processing loads. MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster [14]. It is one of the most popular frameworks to support big data applications on a data center and is widely used by many Internet companies such as Google and Facebook.

We implemented GreenMap, a modified MapReduce framework, on top of a series stack that assigns tasks in synchronization. In this thesis, we analyze and design scheduling algorithms for synchronizing both homogeneous and heterogeneous tasks. We implement the GreenMap scheduling algorithm on a Hadoop framework and construct a small series stack for measurement. The conversion losses are computed based on the measured current profile of each server.

We first evaluated GreenMap with the SWIM benchmark [15]. We found that at all loads, GreenMap achieves 81x-138x reduction in conversion loss from the commercial-grade high voltage converter used by conventional data centers, which is equivalent to 15% reduction in total energy consumption. Even compared to the best available high-voltage converter, GreenMap still achieves a 27x-46x reduction, but the best available converters are not widely used due to high cost. We also evaluate the effect of the GreenMap scheduling algorithm alone by implementing both it and the FIFO scheduling algorithm on serial stack and comparing the conversion loss. This set of experiments shows that with proper selection of scheduler parameters, we can achieve a reduction of 70.1% - 75.4% at all loads by using the GreenMap scheduler.

As GreenMap delays task assignment for synchronization, it suffers an increase in average response time compared to conventional FIFO schedulers; however, the degradation decreases as load goes high. Based on this observation, we further describe the use of GreenMap together with dynamic scaling of data center clusters at the low-load region, which offers a favorable tradeoff between response time and power efficiency.
2.1 Conventional Stack

In the power delivery structure adopted by today’s data centers, the grid power has to go through a set of converters before it is utilized by servers. An example of a DC power distribution map in data centers is given in Figure 2.1

![DC power distribution map](image)

Figure 2.1: A typical of DC power distribution map in data centers

As shown in Figure 2.1, the rectified grid voltage, which is typically 380 V, has to go through a feeder transformer, rectifier, and UPS units before delivery to each of the servers. In each server, there is a DC-DC converter which transforms the high voltage on the DC-bus to a lower voltage to power servers, which typically work at 12 V. Though widely adopted by today’s data centers, the DC-DC converter has several drawbacks, one of which is the large voltage step-down in each DC-DC converters that causes extra power loss, which is inevitable as long as the server is running.

The typical efficiency of a commercial-grade high-voltage converter that is used to realize the large voltage step-down is 80 – 90%. That is, 10 – 20% of the total energy consumption is wasted as conversion loss, which largely limits the system efficiency. Though the efficiency of the best available high-voltage converters could be as high as 95%, they are not widely used due to
high cost and large size [13].

Equation (2.1) shows the computation of conversion loss in the conventional high-voltage converters, which is denoted by $L_{\text{conv}}$. Letting $P$ denote the total power consumption, $E$ the converter efficiency, $V$ the server voltage, $I_i$ the current in server $i$, and $n$ the total number of servers, we have

$$L_{\text{conv}} = (1 - E)P = (1 - E)V \sum_{i=1}^{n} I_i$$

where $V = 12V$,

$$E = \begin{cases} 
0.8 - 0.9 & \text{for commercial-grade converters,} \\
0.95 & \text{for best available converters.}
\end{cases}$$

2.2 Series Stack

Recently, a new power delivery architecture has been proposed in the power community [13]. In this new architecture, the high grid power is first rectified, or converted to a lower DC voltage, typically 48 V DC, and distributed as stack power supply. Servers are grouped into stacks, and in each stack, a set of $n$ servers is connected to stack power supply serially, rather than in parallel as in the conventional systems, as is shown in Figure 2.2. Instead of employing a high-voltage step-down in an AC-DC high-voltage converter for each server, the new architecture achieves an inherent voltage step-down by having the servers evenly sharing the stack voltage; i.e., each server’s input voltage is approximately $1/n$ fraction of the stack voltage. With a suitable choice of $n$, each server shares the correct working voltage. For example, with stack voltage equal to 48 V, and $n = 4$, each server could share approximately 12 V which is the server’s working voltage.

However, since the serially connected servers conduct the same current, their voltage may incur a fluctuation if the equivalent resistance across servers varies over time. Note that there is no converter to regulate the voltage for each individual server, so the input voltage variation of servers may exceed allowable voltage band and as a consequence, servers might crash, or suffer permanent damage. Moreover, a crashed server in a stack is equivalent to an open circuit, and all the other servers will not receive power supply and
thus power out.

To regulate the voltage across servers, [13] introduced bidirectional differential power converters to maintain the allowed voltage band for the servers. The differential converter provides instantaneous mismatch current make-up by extracting current from servers with high equivalent resistance, and injecting current to servers with low equivalent resistance. Since the differential power converters do not need employ high-voltage step-down, the efficiency of the converter could be made as high as 95%, at a reasonable cost and a much smaller size than high-voltage converters.

The following equations demonstrate the conversion loss in differential converters, denoted by $L_{\text{diff}}$. The server voltage $V$ can be considered constant at 12 V due to voltage regulation, so we have

$$L_{\text{diff}} = 1.5(1 - E)V \sum_{i=1}^{n} (I_i - I_{\text{avg}})$$  \hspace{1cm} (2.2)

where $V = 12V$, $E = 0.95$,

$$I_{\text{avg}} = \frac{1}{n} \sum_{i=1}^{n} I_i.$$  

The equations show that the conversion loss in differential converters is proportional to the difference in servers’ currents, which is caused by mismatch of work load (equivalently power consumption) between servers. The extra factor of 1.5 is due to the specific topology of the server-to-virtual-bus differential converter [13]. The topology is shown in Figure 2.3, where each differential converter (DC-DC converter) connects the corresponding server to a virtual power bus, and moves current between them.

We can integrate series stack structure into data center racks. As illustrated in Figure 2.4, a rack can consist of more than one series stack. This facilitates the installation of a series-connected stack and provides proper ground isolation [13]. Note that the server hosting the resource manager (RM) is outside the series stacks, since its computational load is very different from the other servers, and thus is ill suited to be balanced with other servers. We also allow the combination of series stacks and conventional servers in a data center, which provides more scalability and flexibility.
2.3 MapReduce and Hadoop

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster [14]. It is one of the most popular frameworks to support big data applications on a data center and is widely used by many Internet companies such as Google and Facebook.

Under the MapReduce model, users are asked to submit a data processing job. MapReduce jobs are usually split into multiple data processing tasks and launched on many machines in a cluster. “Job” is a high level concept, closely defining the need of a user’s computation request, whereas task is a low level concept, which specifically defines the concrete data and computation resource that are needed from the underlying system.

A MapReduce job generally contains three main stages: Map, Shuffle Reduce. In Map stage, the data processing job is split into many small data processing tasks, usually according to the data stored to each machine. Each task is deployed on one machine and produces a partial result based on the machine’s local data in the format of ⟨key-value⟩ pairs. When Map tasks are finished, we would like to summarize the partial result of all Map tasks. In Shuffle stage, the partial result of each map task is re-delivered to another machine based on their keys, such that all ⟨key-value⟩ pairs of the same key go to the same machine. In Reduce stage, all machines will summarize the partial sent to it and yield the final result.
Figure 2.5 is an example of a MapReduce word-count job. It counts the number of occurrence of each word in a string. (Note that the string can be very long, and split into data chunks and stored on multiple machines.)

The fastest way to complete a job is to divide the computation request in a balanced manner such that the running times of all tasks are approximately the same in Map or Reduce stage. With this design, a MapReduce task can potentially reduce a significant amount of extra power loss from deployment of series stack. This is because when a MapReduce job is carefully split into tasks of similar size, all machines in a stack will run for approximately the same amount of time and yield almost zero load mismatch. This minimizes extra power loss in converters.

The MapReduce jobs are hosted by an underlying distributed system: Hadoop. The general structure of Hadoop is shown in Figure 2.6. Generally speaking, the underlying Hadoop structure can be divided into master and slave components. The slave component usually consists of multiple slave servers. Each slave server stores data on the Hadoop distributed file system (HDFS) and performs computation tasks. The master component usually contains one master server in a small data center, and it manages all the jobs in the system, including receiving jobs from users, managing job progress, and replying to users with job results. The master server also coordinates
the work of all the slaves, including assigning tasks, monitoring task status and monitoring the health of all slave servers. From the job-task perspective, the master server is also known as the job tracker, and slave servers are task trackers. Between master component and slave component, there are multiple types of communication methods for slaves to share information to the master, and for the master to distribute commands to slaves.
Figure 2.6: Hadoop structure
As is demonstrated in Chapter 2, the conversion loss in differential converters is proportional to imbalance of workload across the servers. In order to reduce the conversion loss in a series stack, we need to balance servers’ load in real time.

3.1 Current Profiling

We start by profiling the power consumption of a word-count job containing one map task and one reduce task on a server with fixed voltage as 12 V, as is shown in Figure 3.1. The current consumption is measured by a Yokogawa wt310 digital power meter.

![Figure 3.1: Current consumption of a word-count job with one map task and one reduce task](image)

The idle current is around 2.8 A. The setup task initializes the job and...
creates temporary output directories, consuming close-to-peak current at 5.5 A for 2.5 s. The server goes idle for another 2.5 s before launching the map task. The beginning of the map task consumes close-to-peak current as a new thread is initialized and data are read into memory. However, the bulk of the map task experiences an oscillation of current around 4.8 A, as it generates <key,value> pairs and outputs them to the intermediate directory. The alternating computation-intensive and I/O-intensive operations cause the current to oscillate. The beginning of the reduce task also consumes close-to-peak current as a new thread is initialized, followed by 4 seconds of low current at 2.8 A, as <key,value> pairs are copied from intermediate directories on other servers. The later stage of the reduce task is characterized by large oscillations between 2.8 A and 5.5 A due to the intersection of high-current computation-intensive operations and the low-current I/O operations. The cleanup task after the job’s completion causes another short period of close-to-peak current consumption.

In general, a MapReduce job always has a setup task and a cleanup task. It can have multiple map tasks and reduce tasks, whose current consumption can vary depending on user-defined functions, although map tasks (or reduce tasks) of the same job will still have similar current profiles.

3.2 Synchronized Assignment for Homogeneous Task

We built GreenMap to balance the computational loads for series stack by synchronizing task assignment. In this this section, we are only interested in the most basic MapReduce workload, where each job only contains map tasks, and all the map tasks are identical. This assumption is valuable because some data centers perform only one data processing task, meaning all the MapReduce jobs are homogeneous. Given homogeneous jobs, we can always divide the input data of MapReduce tasks into equally sized data chunks, to make all tasks of equal size. There are three main modifications to the default MapReduce scheduler.

First, the setup and cleanup tasks are moved to the server residing Resource Manager. As each setup (and cleanup) task is executed only once per job, and it consumes close-to-peak power, it is inherently unsuitable for parallelization and balancing across a series-stack of servers. In a more scalable
implementation, they can be assigned to any conventional servers outside series stacks.

Second, we minimize load imbalance by assigning the same number of map tasks to each server, i.e. synchronizing assignment, and whenever possible, assigning map tasks from the same job in synchronization. This is achieved by delaying task assignment until the number of outstanding tasks is at least that of the servers with idle slots. In particular, when all \( n \) servers in a stack have idle slots and the number of outstanding tasks exceeds that of servers, a batch of \( n \) tasks are assigned in synchronization, one per server, in accordance with the assignment by the default FIFO scheduler. Otherwise all outstanding tasks are delayed. In the latter case, the number of servers with idle slots increases over time as running tasks are completed, and the number of outstanding tasks also grows as new jobs arrive. Eventually, the conditions of the former case will be satisfied, and assignments will go on.

Third, a timeout mechanism is used to prevent the system from delaying tasks for too long. More specifically, a timer is set to zero whenever tasks are assigned in synchronization or a new job has arrived. In the absence of neither, when the timer reaches a threshold value, all outstanding tasks are assigned. A larger value for the threshold will further reduce power conversion loss and increase response time, while a smaller value will increase power loss and reduce response time.

In order to show the effect of task synchronization, we ran a small demonstrative trace on four servers under conventional power delivery structure, and compared the measured current profiles. The trace consists of two jobs of 1 map task, one job of 2 map tasks and one job of 8 map tasks, arriving at random intervals. Figure 3.2(a) shows the measured current profiles of the four servers with no task synchronization. Not surprisingly, we observe a large difference in currents consumed at each server, which implies the unbalanced load across the servers. Figure 3.2(b) shows the current profiles of the same four servers with synchronized task assignment. We observe that the map tasks are indeed synchronized, and the difference in currents consumed at different servers becomes small and unusual.
3.3 Synchronized Assignment for Heterogeneous Tasks

Section 3.3 provides an algorithm for assigning homogeneous Map tasks synchronously. In reality, the actual running time of a MapReduce task may vary depending on multiple factors: type of MapReduce job, data fetching speed, system congestion, etc. So it is also valuable to consider heterogeneous tasks, i.e. tasks of different running time.

The ideal case is that we know how many tasks are coming and how long each task will run in the future. However, the running time of a given task may differ based on the temporal status of a data center and thus is difficult to precisely predict. Some works [16] proposed a machine-learning based method to predict future load in a data center based on historical statistical data, which may provide some hint of the running time of each task. In this section, we simply assume that there is little information about each task and data center, and that the running time of each task can only be revealed when it is actually finished.

The key idea behind heterogeneous task assignment is mostly similar to that for homogeneous tasks: generally speaking, when the number of task is sufficient, we tend to assign the same number of tasks to each machine in the system. To prevent tasks from waiting forever, we set a timeout such that when a task has been waiting in the system for too long, it must be assigned immediately when a computation slot is available.

Note that the power loss for series stack depends on the imbalance of the system occupancy. Assume a system of four servers, each with one slot. Let \( V \) be the working voltage for every server, and \( I \) be current when a server is occupied. When 2 out of the 4 servers are occupied, the average current is \( \frac{I}{2} \), and the total power loss is \( C \times (\frac{|I/2|}{2} + \frac{|I/2|}{2} + \frac{|I/2|}{2} + \frac{|I/2|}{2}) = 2C \times I \), where
\[
C = (1 - 0.95) \times V.
\]
And when 3 out of 4 servers are occupied, the average current is \( \frac{3I}{4} \), and the total power loss is \( C \times (\frac{I/4}{4} + \frac{I/4}{4} + \frac{I/4}{4} + \frac{|-3I/4|}{4}) = \frac{3}{2}C \times I \).

It means if there are 2 tasks assigned in the system due to timeout, a smarter way to further reduce extra power loss in series stack is to simply lunch another waiting task.

The detailed algorithm is as follows:
Algorithm 1  Synchronous Task Assignment

1:  procedure SCHEDULING
2:     system parameter: timeout
3:     Sort task in task_queue by FIFO order
4:     for task in task_queue do
5:         if current_time − task.enter_time ≥ timeout then
6:             if system.slot > 0 then
7:                 assign(task)
8:         end if
9:     end if
10:    end for
11:    if system.slot < number of tasks in task_queue then
12:       for task in task_queue do
13:          if system.slot > 0 then
14:             assign(task)
15:       end if
16:    end for
17:    end if
18:    if system.slot == 3 and task_queue is not empty then
19:       let task = task_queue.first_task
20:      if system.slot > 0 then
21:         assign(task)
22:    end if
23:    end if
24:  end procedure
25:  procedure ASSIGN(task)
26:     system.assign(task)
27:     system.slot = system.slot − 1
28:     task_queue.remove(task)
29:  end procedure
Figure 3.2: Current profiles of four servers

(a) Imbalanced loads with no task synchronization

(b) Balanced loads with task synchronization
CHAPTER 4
GREENMAP: IMPLEMENTATION

We integrate GreenMap scheduler into MapReduce structure and deploy it on a series stack consisting of 4 servers. This chapter demonstrates the implementation details of GreenMap scheduler and how it affect data center behaviors.

4.1 General Structure: Master-Slave

As is shown in section 2.3, we have introduced the general master-slave structure of the MapReduce-Hadoop framework. To implement GreenMap under MapReduce structure, the key is for the master to synchronize tasks assignment to slaves. To achieve synchronized assignment, we need to complete the following two tasks: 1 master must understand the system occupancy, e.g. how many slaves are occupied. 2 master must be able to make scheduling decisions based on system occupancy and current waiting tasks and assign task to slaves in a synchronized manner. In order to achieve the above goals, we modify two key components in the MapReduce structure: Heart Beat Protocol and MapReduce Scheduler

4.2 Heartbeat Protocol

A heartbeat is a signal indicating that one is alive. The heartbeat in MapReduce framework is just like the heartbeat for a human being. Every slave server sends heartbeats to master server periodically, indicating that it is still working correctly. If the master server did not receive any heartbeat of a slave server in a period of time, then it may think that the slave is dead. When a slave is dead, the master server will handle any failure events, such as unfinished tasks and unavailable data, to make sure that the user gets
a correct response: either the overall process is still running correctly and continues or it aborts due to errors.

Except for a simple acknowledgment, the heartbeat may also contain extra information that can be used as the basics for the scheduling algorithm. For example, the heartbeat always carries the occupancy of a server, indicating how many slots are available on the server. For this preliminary work, the synchronized task assignment algorithm only needs the number of available slots. Future implementations may consider piggyback extra information such as CPU utilization rate.

The heartbeats are sent periodically with a default interval to be 1 second. This is small compared to the general length of a Map or a Reduce task, which is usually more than 1 minute. However, in order to more precisely synchronize tasks in a system, we change the interval to 300 ms. Comparing to normal task length, this interval allows the master server to have an almost real-time view of the system.

4.3 Scheduler Implementation

With heartbeat from slave servers, the master server has a global view of the entire stack, which is the basic information needed for the master server to make decisions. The default Hadoop scheduler is a first-in-first-out scheduler (FIFO scheduler), which assigns the first available slot immediately to the task that is submitted first. The task assignment is achieved by remote procedure call (RPC), which allows the master server to send command to slave servers by calling a local function. This function will pass necessary information of a task, including the specific task description and the location of input data to the assigned slave server. This information will be capsuled to an Internet package and carried by the RPC protocol until it is received by the target slave server. This slave server will further dispatch the package and execute computation following the description of the task.

In order to implement the GreenMap scheduling algorithm, we modify the FIFO scheduler. Instead of assigning tasks immediately when there is an available slot, we hold tasks until the number of waiting tasks in the system is sufficient for all slave servers to get the same number of slots occupied. We then distribute the tasks to all servers by calling the task assignment
function to send an RPC to the corresponding slave server.

Note that even after assigning $n$ tasks to a slave server, if no future heartbeat is received from the slave, the master may still think that the number of available slots in this slave is unchanged, and thus may send extra tasks to the slave, which may potentially create imbalance of load. To guarantee that a slot is not doubly used, we only make a decision when the heartbeats of every slave server are received. Based on the 300 ms heartbeat interval we set, the waiting time for collecting all slave heartbeats is at most 300 ms, which still guarantees the task assignment is almost real-time.
CHAPTER 5

EVALUATION

5.1 Experiment with Homogeneous Tasks

5.1.1 Experiment Setup

Our test bed includes five Dell Optiplex SX775 Core 2 Duo workstations, each containing two available slots. One server hosts the Resource Manager (RM) and is not in a series-stack. The remaining four servers simulate a series-stack of 48 V.

We connect the four servers with a conventional power delivery architecture, and measure the power consumption of each server using a Yokogawa wt310 digital power meter with 10 samples per second per server. We compute the power conversion loss using Equations (2.1) and (2.2). The advantage of this setup is that it allows us to compare the conventional conversion loss and the differential conversion loss in the exact same setting, with the same run of a trace.

The setup is shown in Figure 5.1. On the left side, five servers form a Hadoop stack consisting of one master server and 4 slave servers. The five servers are connected to a local area network such that they can communicate with each other. In the middle are 4 Yokogawa wt310 digital power meters where each one of them measures the current and power information of one slave server. Every server is connected to the power supply via the switch boxes on the top, which delivers the same amount of voltage and current through the measurement equipment. The four pieces of measurement equipment measure 10 data points of voltage and current per second, and deliver the result to the server on the right side which is connected to all four pieces of equipment.
5.1.2 Workload Model

Abad et al. [17] studied the statistical features of the MapReduce workload, which indicates that the amount of computation requirement per job for a system follows a Pareto distribution, which is a heavy tail distribution. We generate traces by selecting jobs from the SWIM benchmark [15] so that we achieve a good representation of the Pareto job size distribution [17], and the length of the trace and the number of files are appropriately scaled for the capacity of one series-stack.

Table 5.1: Job size distribution

<table>
<thead>
<tr>
<th>Bins</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job count</td>
<td>25</td>
<td>9</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Map count per job</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>100</td>
</tr>
</tbody>
</table>

Job arrivals are generated as a Poisson process, and the jobs contain no reduce tasks. The data block size is set to 32 MB, and each map task takes an average of 70 seconds. Hence for each load point, the trace takes $1.5 - 6$ hours on our cluster. After scaling, our trace contains 447 tasks and 50 jobs. Table 5.1 shows the job size distribution.

The experiments are conducted at various loads. In a cluster, the com-
putational load is computed by:

\[
Load = \frac{N}{S} \times \frac{T_{\text{complete}}}{T_{\text{arrival}}}
\]  \hspace{1cm} (5.1)

where \(N\) is the average number of tasks in a job, \(S\) denotes the total number of slots in a data center, \(T_{\text{complete}}\) denotes the expected completion time of a task, and \(T_{\text{arrival}}\) means the expected interval between two job arrivals.

5.1.3 Experiment Result

Figure 5.2: GreenMap reduces power conversion loss from the conventional architecture by two orders of magnitude

Figure 5.2 shows that at all loads, GreenMap achieves 81x-138x reduction in conversion loss from the conventional power delivery with a commercial-grade high voltage converter of 85% efficiency, which is typical of converters used in data centers today. The power conversion loss is reduced by two orders of magnitude, from an average of 31.4 W to 0.3 W. This is equivalent to 14.999% reduction in total energy consumption, almost eliminating the 15% conversion loss altogether.

Figure 5.2 also shows the conversion loss of the conventional power delivery with the best available high-voltage converter of 95% efficiency. GreenMap
Figure 5.3: GreenMap achieves response time comparable to that of Hadoop FIFO scheduler at load 0.6 and above, while increasing response time at lower loads achieves 27x-46x reduction in power conversion loss, from an average of 10.45 W to 0.3 W.

Figure 5.3 shows the average job response time of the default Hadoop FIFO scheduler versus that of GreenMap. As GreenMap delays task assignment until tasks can be assigned in synchronization, the average response time below 0.6 load increases by 26 – 42%. However, when the load reaches 0.6 and above, no degradation in response time is observed. This is because there are an abundance of outstanding tasks at high loads, and tasks are seldom delayed, whereas at low loads, sparse task arrival makes delay more common.

Also note that in Figure 5.3, as computational load grows from low to medium, the increase in average job response time of GreenMap is smaller than that of the conventional FIFO scheduler. For instance, the average response time of GreenMap increases by 41% when load grows from 0.3 to 0.7; however, that of the conventional FIFO scheduler increases by as much as 100%.
5.2 Experiment with Heterogeneous Tasks

This section demonstrates the performance of GreenMap with heterogeneous tasks. We ran simulation of 1000 tasks generated by Pareto distribution. The task distribution is shown in Figure 5.4. We compare the performance of naive FIFO scheduler and GreenMap scheduler with different tolerance of task waiting time.

![Job Running Time Statistics](image)

Figure 5.4: The statistics of job size with Pareto distribution

Figure 5.5 shows the total power loss caused by serial stack during the simulation at different levels of load. We can see that in general the extra power loss decreases as the load increases. This is because at higher load, all machines are occupied in the most time, meaning the load is mostly balanced and less power is reduced. It can be observed that conversion power loss is reduced by 33.4% - 70.1% at low load and by 48.9% - 73.9% at high load when GreenMap is deployed. With task waiting time up to 15 minutes, the power loss at all loads is reduced by 70.1% - 75.4%. With a smaller tolerance of task delay, we can also achieve as much as 52.2% power saving at load higher than 0.6. The reason that we can easily achieve power saving at high load is that there are always sufficient tasks occupying the system, and even if the number of waiting tasks is not enough to occupy all four machines, the
next task is expected to arrive soon.

Also note that once the system load is given, if the timeout is not big enough, we cannot achieve much improvement, because power loss appears when the number of tasks in the system is not enough for assigning synchronously to all machines, and when the tolerable task delay is not enough for the tasks in the system to wait for the next task, the power loss will still occur. On the other hand, when a significant power reduction has been reached, further increase in task waiting time may not be worthwhile. For example, at load larger than 0.6, there is no significant difference between 10 minutes waiting time and 15 minutes waiting time.

Figure 5.6 shows the effect of GreenMap on task response time. At lower load, the increment in response time is almost linear to the task waiting time. GreenMap with 5 minutes task waiting time increases response time by 3.6 minutes, and GreenMap with 10 minutes task waiting time increases response time by 6.2 minutes. With task waiting time of 15 minutes, the task response time increases by 9.5 minutes. At higher load, the response time is 0.8 - 10.2 minutes, corresponding to 18% to 224% increment. As we can see, the increase in response time is similar across all loads, which means that most tasks need to wait as long as possible to be able to get a synchronized
From Figures 5.5 - 5.6, we can see that as the waiting time of GreenMap increases, the response time increases almost linearly, and extra power loss decreases in a sub-linear manner. This indicates that one should carefully choose the waiting time based on the sensitivity of tasks in a system. When a system is sensitive, the waiting time should just be able to meet the average response time requirement of the service objectives of a specific cloud system, so that we can achieve the best reduction in power loss; when a system is delay-tolerant, we should trade off between smaller waiting time, which yields better response time and reasonable power saving, and a larger waiting time, which saves as much extra power as possible.

5.3 GreenMap with Dynamic Scaling

As is shown by the above result, GreenMap suffers a degradation in response time when the load is below 0.6, and as load grows from low to medium, the degradation decreases. In fact, the lower the load, the more delay GreenMap will apply to tasks in order to emulate a higher load, at which point
there is an abundance of outstanding tasks, hence facilitating assignment in synchronization. We also observe that higher loads can be more efficiently achieved by turning off a fraction of stacks in a large cluster with multiple series-stacks. The preference of higher load by GreenMap provides us with inspirations for further improvement: When the cluster is running at a low load, one can consolidate services to only a fraction of servers and have them running at a higher load, while on the other hand, saving energy from other idle servers by shutting them down.

For instance, assume 10 series-stacks are running at 0.4 load. From Figure 5.2, the total power consumption in each series-stack of 4 servers is 192.2 W ($= V \sum_{i=1}^{4} I_i$) at 0.4 load, and the conversion loss in each series-stack is 0.29 W with GreenMap. With dynamic scaling, we can turn off 3 series-stacks, resulting in 0.57 load for each remaining series-stack. The corresponding power consumption in each series-stack is now 215.4 W, and the conversion loss is 0.33 W with GreenMap. Hence, with GreenMap but not dynamic scaling,

$$\text{total power} = (192.2 + 0.29) \times 10 = 1924.9\text{W},$$

whereas with GreenMap and dynamic scaling,

$$\text{total power} = (215.4 + 0.33) \times 6 = 1294.4\text{W},$$

which is a 32.8% reduction. The reduction in total energy consumption is similar as the servers are mostly idle at 0.4 and 0.57 load, and the trace takes a similar amount of time to finish. The average job response time with GreenMap will increase by only 15% as the load increases from 0.4 to 0.57, yielding a favorable tradeoff between power efficiency and response time.
CHAPTER 6

CONCLUSION

We explored the feasibility of series-connected stacks in data centers by implementing GreenMap, a modified MapReduce framework that assigns tasks in synchronization. We found that with task synchronization, the conversion loss in data centers can potentially be reduced by two orders of magnitude, which is equivalent to about 15% of total energy consumption. GreenMap also achieves 70.1% - 75.4% reduction in conversion loss compared to Hadoop’s FIFO scheduler under serial stack structure. Based on the observation that the average response time of GreenMap suffers a degradation at low load, we further proposed a modification of GreenMap with dynamic scaling to achieve a favorable tradeoff between response time and power efficiency. Future work includes implementing GreenMap with multiple series-stacks and heterogeneous jobs, and evaluating the system on actual series-connected stacks.
REFERENCES


