Towards Energy Efficient MapReduce

Yanpei Chen
Laura Keys
Randy H. Katz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-109
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-109.html

August 5, 2009

Towards Energy Efficient MapReduce

Yanpei Chen, Laura Keys, Randy H. Katz
RAD Lab, EECS Dept. UC Berkeley
{ychen2, laurak, randy}@eecs.berkeley.edu

ABSTRACT

Energy considerations are important for Internet datacenters operators, and MapReduce is a common Internet datacenter application. In this work, we use the energy efficiency of MapReduce as a new perspective for increasing Internet datacenter productivity. We offer a framework to analyze software energy efficiency in general, and MapReduce energy efficiency in particular. We characterize the performance of the Hadoop implementation of MapReduce under different workloads. We also introduce quantitative models to guide operators and developers in improving the performance of MapReduce/Hadoop. A major, but somewhat unsurprising finding is that for workloads where the work rate is proportional to the amount of resources used, improving the performance as measured by traditional metrics such as job duration is equivalent to improving the performance as measured by lower energy consumed.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Design, Performance, Economics

Keywords

MapReduce, Hadoop, Energy, Data centers

1. INTRODUCTION

The energy consumption of Internet datacenters is an important aspect of datacenter efficiency. Research has shown that over the lifetime of IT equipment, the operating energy and cooling cost is now comparable to the initial equipment acquisition cost [13]. Also, most operators build datacenters to be as large as they can be economically supported by the power grid. Thus, equipment that is not power or energy efficient will limit the amount of IT equipment that can go into a particular physical datacenter site. As computational needs increase, this means more frequent, large scale investments in new datacenter sites.

At the same time, the total power consumption of datacenters in the United States has doubled from 2000 to 2005 representing 1.5% of the national power consumption in 2005, larger than that of color televisions [19]. During the same period and thereafter, the national power production level witnessed only marginal increases [1, 8], a sharp contrast with the 15% yearly increase in datacenter power consumption [19]. Thus, datacenter operators, with their long history of innovation, have a technology responsibility to lead the nation in using new ideas to reduce our carbon footprint.

Datacenter operators have already taken on this responsibility, and have developed industry power efficiency standards such as the power utilization efficiency (PUE) metric [4]. The PUE metric is the ratio of total facility power and total IT equipment power. In 2004, typical PUE values are 2 and above [21], suggesting that most of the datacenter power is wasted in physical, non-IT systems such as cooling and power distribution. More recently, PUE values have improved significantly, and are fast approaching the ideal value of 1 [18, 22]. This means that the vast majority of datacenter power consumption is now in the IT equipment. Consequently, the greatest opportunities for further improvement now lie in optimizing IT.

We can improve the energy efficiency of IT equipment through either hardware or software. We focus on software because we believe it presents a lower barrier to innovation and adoption. Of all the software paradigms, we analyze MapReduce [15] because it powers the core business of many key Internet enterprises. Of all the MapReduce implementations, we choose Hadoop [6] because it is open source, and thus offers a low barrier for innovation. We suspect there will be many opportunities to save energy, because existing Hadoop versions have not been designed with energy efficiency in mind.

To the best of our knowledge, our work is the first detailed study of the energy efficiency of datacenter applications like MapReduce. We seek to make several contributions. We present an analytical framework for software energy efficiency, including efficiency of systems such as MapReduce. We provide experimental data to characterize the energy consumption for the Hadoop implementation of MapReduce. We also introduce quantitative models that can help Hadoop operators and developers in identifying opportunities to improve energy efficiency.

This report is organized as follows. Section 2 gives a brief overview of MapReduce, and quickly review related work in Internet datacenter energy efficiency. Section 3 presents our software energy efficiency framework, and discusses the complex inter-relationship between metrics, system parameters, workloads, and energy measurement. Section 4 describes our experimental setup and our energy measurement methodology. Section 5 provides descriptive data to characterize the energy consumption of several MapReduce workloads. Section 6 outlines quantitative models that allow Hadoop operators to tune their clusters, and Hadoop developers to identify potential improvements for the Hadoop system. Section 7 summarizes our recommendations, and suggests directions for future work.

2. BACKGROUND

2.1 MapReduce Overview

MapReduce was initially developed at Google as a platform to parallel processing of large datasets [15]. Today, MapReduce powers Google's flagship web search service, as well as clusters at companies like Yahoo!, Facebook, and IBM [7]. MapReduce has several attractive qualities. Programs written using the MapReduce model are automatically executed in a parallelized fashion on the cluster. Also, MapReduce can be run on clusters of cheap commodity machines, an increasingly attractive alternative to expensive, specialist clusters. Furthermore, MapReduce is highly scalable, allowing petabytes of data to be processed on thousands and even millions of machines. Most importantly for our work, MapReduce is a data intensive benchmark that generates a mixture of CPU, disk, and network activity that is representative of many real-life datacenter workloads.

At its core, MapReduce has two user-defined functions. The Map function takes in a key-value pair, and generates a set of intermediate key-value pairs. The Reduce function takes in all intermediate pairs associated with a particular key, and emits a final set of key-value pairs. Both the input pairs to Map and the output pairs of Reduce are placed in an underlying distributed file system (DFS). The run-time system takes care of retrieving from and outputting to the DFS, partitioning the data, scheduling parallel execution, coordinating network communication, and handling machine failures.

A MapReduce task is executed in several stages. There is a master daemon responsible for coordinating a job on a cluster of workers. The input data resides in the DFS. The master divides the input file into many splits, each to be read and processed by a Map worker. The intermediate key-value pairs are periodically written to the local disk at the Map workers, usually separate machines from the master, and the locations of the pairs are sent to the master. The master forwards these locations to the Reduce workers, who read the intermediate pairs from Map workers using remote procedure call (RPC). After a Reduce worker has read all the intermediate pairs, it sorts the data by the intermediate key, applies the Reduce function, and finally appends the output pairs to a final output file for the Reduce partition. The output file also resides in the DFS. If any of the Map or Reduce tasks lags far behind, backup tasks are executed speculatively, and the task is consider complete when any of the backup Map or Reduce workers finishes. To limit network traffic, users may additionally specify a Combine function that merges multiple intermediate key-value pairs before sending them off to the Reduce worker.

For our work, we select the Hadoop implementation of MapReduce [6]. This is one of the most popular MapReduce implementations. It is an open source Java application. The user-supplied Map and Reduce functions are both written in Java; the cluster configurations are set using XML. Extensive documentation exists online. The Hadoop DFS (HDFS) implements many features of the Google DFS [17], on which the original Google MapReduce runs.

2.2 Related Work

There has been a significant amount of prior work in energy efficient computing systems. First motivated by mobile applications, the interest in energy efficiency has been sustained by the proliferation of laptops, as well as the recent focus on Internet datacenters. It would be impossible to comprehensively review the existing literature, so we will highlight below several topics most closely related to our work.

Frequency scaling has long been implemented as a well known power saving technique in CPU architecture. However, with CPU frequencies scaled down, any CPU-bound workloads would take proportionally longer to run. Since total energy use is the product of power and time, a lower frequency actually impose a latency cost at no benefit in terms of total energy use. Nevertheless, frequency scaling is still useful if the workload is not CPU-bound, or if power is a system design limitation, as it is often the case in the CPU design space. The higher the CPU power, the greater its cooling needs. The now well-known tradeoff between power and latency in frequency scaling highlights for us the importance to look at both energy, and power, i.e., the rate of energy consumption.

In a different line of work, energy conscious communications for mobile architectures developed the energy per useful bit metric [11]. This metric highlighted the need for metrics that evaluate systems with regard to the energy used for each unit of useful work. In a similar fashion, we can also measure finishing time per unit of useful work. It is difficult, if not impossible, to come up with a general definition for "useful work". But for comparable workloads, the unit of useful work is usually easy to identify. For example, in HDFS read or write jobs, the unit of useful work would be the bytes read or written, and the corresponding workload-conscious energy efficiency metric would be energy per bytes read or energy per bytes written.

More recently, people have started developing benchmarks to measure datacenter energy efficiency. The SPEC Power benchmark [10] has been widely accepted by server vendors. As already mentioned, the PUE metric [4] is already a defacto industry standard among datacenter operators. The EPA has also developed a new set of Energy Star criteria for enterprise servers [2]. A set of Energy Star criteria for enterprise storage is under development as well [3]. The natural progression of these workload-specific energy metrics would be some sort of unifying datacenter productivity measure. Our work offers insights that help guide incremental steps towards that goal.

Also, people have recently started speaking about energy proportional computing [12] as a worthy system design goal. The main argument is that current systems consume nearly as much power idle as when they are actively doing work. Given that most machines are expected to see a significant amount of idleness during their lifetime, we should design computing systems such that they consume power proportional to the amount to which they are utilized. Ideally, an energy proportional system consumes zero power when idle, full power when under 100% utilization, and the power increases linearly between the two end points. The energy proportionality concept inspired the quantitative analysis leading to our major result that energy efficiency and traditional time efficiency are fundamentally equivalent.

Although we are the first to look at the energy efficiency of MapReduce, we are certainly not the first to use MapReduce in work on systems energy consumption. MapReduce has already become a standard workload in evaluating the power consumption of datacenters [16, 20]. We fully believe that MapReduce should be a part of any evaluation of datacenter efficiency. We differ from this line of work by focusing on improving the software energy efficiency of MapReduce in particular, instead of using MapReduce as a workload for a wider energy efficiency investigation.

Lastly, in a closely related work, JouleSort investigates the energy efficiency of sorting applications [23]. Sorting is a key part of the MapReduce system. In most implementations, including Hadoop, sorting occurs implicitly with every MapReduce job. The JouleSort performance metric of joules per sorted record echoes the energy per useful bit concept. The JouleSort experimental methodology also informed ours. Their key finding is that sorting is most energy efficient when it is CPU bound instead of IO bound. The insight in this area would help us improve the sorting part of MapReduce. Our investigations are a superset of the focus in JouleSort. JouleSort is one of several workloads that we run on MapReduce.

3. ANALYTICAL FRAMEWORK

We believe software energy efficiency studies should be guided by four considerations - the desired performance metrics, the available system parameters, the appropriate workload, and the method of energy measurement.

3.1 Metrics

The choice of performance metrics is critical to any discussions about datacenter energy efficiency. First, we must make a distinction between energy and power. Power is the rate of energy consumption, i.e. energy = power × time. Datacenter operators are charged for energy consumption, usually in kilo-Watt-hours (kWHr). However, each datacenter site is usually constructed for a certain power rating, in kilo-Watts (kW) or even mega-Watts (MW). For a constant power, jobs that run longer consume more energy. For jobs that consume a fixed amount of energy, a faster finishing time would come at the cost of higher power. Both metrics are important in a discussion about datacenter energy efficiency.

Another crucial metric is energy per unit of useful work. This metric is important because it gives us a way of talking about energy consumption independent of workload size. The difficulty in using this metric, however, is identifying the appropriate unit of useful work. Doing so is difficult across different types of workloads. However, for workloads of the same type, the appropriate unit of useful work is often obvious, e.g. joules per record sorted for sort, joules per byte written/read for write/read jobs, and the like.

A similar metric is work rate per Watt. While energy per unit of useful work measures the energy efficiency indepedent of workload size, work rate per Watt can measure the power efficiency independent of workload size.

Also, improved energy efficiency should not come at the cost of decreased performance in the traditional sense. Thus, metrics like job finishing time and service level agreements (SLA) remain relevant. We believe that energy efficiency should be an essential part of system performance, and energy efficiency metrics should complement traditional performance metrics, instead of completely replacing them. The focus should not be about whether job finishing time or energy consumption is more important. Rather, we should be asking questions like whether we can use additional energy or power to buy improved finishing time or better SLA, or whether we can conserve energy or power without harming finishing time or SLA. As we will discover later, for most systems, decreasing energy consumption is equivalent to decreasing the finishing time.

To summarize, we believe that a bundle of metrics should be used, because we are not yet at the stage where we can devise a single metric to unify the different system design priorities. We begin by analyzing metrics that are the easiest to collect - finishing time, energy, power, and energy per unit of work.

3.2 Parameters

In talking about the energy efficiency of software systems like MapReduce, it is also important to identify the parameters of the systems that we can change. We believe there are several different types of parameters that we can optimize.

The easiest parameters to change are static software configuration parameters. For MapReduce, such parameters include the number of workers and HDFS data nodes, as well as the many Hadoop-specific parameters found in the hadoop-default.xml. We believe the most interesting parameters are IO parameters including HDFS block size, and HDFS replication factor. They directly impact the performance of the IO path. Since MapReduce is a data-centric paradigm, a sub-optimal IO path is likely to impact all performance metrics for all workloads.

We must add a caveat that a key function of HDFS is to ensure fault tolerance. Changing the HDFS replication factor would change the resistance to faults, and decreasing HDFS replication factor to 1 implies that the failure of any nodes would result in data loss. Thus, an interesting question is how much energy we can save if we are willing to compromise on data persistence. We provide data to quantify this trade-off.

Another set of parameters are dynamic software execution parameters. For MapReduce, such parameters include the different number of map and reduce jobs, different scheduling algorithms, different HDFS block placement schemes, different speculative execution models. Aside from the number of map and reduce jobs, these parameters are usually directly built into the software architecture. Optimizing them often require more extensive code modifications. We do not implement or evaluate any software execution optimizations. However, our findings suggest several opportunities where such optimizations have good potential.

Last but not least, software requires hardware to run, thus hardware parameters are relevant. One can object that hardware is difficult to change, but we have seen that datacenter operators can and do customize their server hardware [24]. One can further object that hardware configurations are not a part of the software optimization problem, but we can respond that different types of software may have different CPU/IO tradeoffs, or are biased towards different types of IOs. Thus, understanding the software-hardware interaction is vital to understanding software energy efficiency. We believe the most important parameters are the choice of CPU and the speed of various IO paths, including disk, memory, network, and the size of memory. We are yet to gather data for different hardware.

3.3 Workload

There are two properties of a good workload for measuring datacenter software energy efficiency. First, the workload should be representative of actual production workloads. Second, the workload should exercise all parts of the system. In particular, the workload should exercise all major parts of the IO path, all common aspects of various scheduling/placement algorithms, and all common combinations of IO/CPU biases. Given these criteria, It is unlikely that a single type of workload can meet all the criteria, and a bundle of workloads would be more realistic. MapReduce is a good starting point because it is representative of large, data-intensive computations. As outlined in Section 2.1, MapReduce powers production clusters at many leading datacenter operators. It also exercises all parts of the computation system by generating a mixture of CPU, disk, and network loads.

The most realistic insights would come from measuring production workloads on production clusters. Such measurements would reflect the effects real-life utilization cycles, unpredictable failures, background workloads, adminstration interruptions, and the like. However, to understand the effects of various system design parameters, we believe we should first perform measurements in a controlled and relatively simple environment. In particular, the background CPU, disk, network loads, and the level of machine virtualization should all be controlled. The simplest environment would exercise the workload with no background load and no virtualization. In such an environment, the effect of software system design parameters would not be distorted by complexities in the experimental setup.

3.4 Energy Measurement

In measuring the software energy consumption, an important task is to define the system boundary. There are various granularities at which we can draw boundaries for the system. We believe the most useful and the most convenient boundary is at the wall power socket. This way, the power-energy measurements would capture the holistic system performance, including any components that may be idle and drawing wasted power. In particular, we can measure the baseline energy consumption by leaving the system on but idle.

All MapReduce master and workers should be measured. Ideally, the network switch should be measured as well. However, monitoring the network switch on a shared cluster exposes our readings to interference from other network intensive jobs on the cluster. Therefore, we suggest that unless the cluster can be isolated, we should not monitor the network switch. This is purely prioritizing the quality of data over the comprehensiveness of data.

As we will discuss in more detail later, including the master in power energy measurements distorts data trends in small clusters. We have only a small cluster of one master and up to twelve workers. Thus, for most of our experiments, we will in fact measure the energy consumed by workers only. Again, this is done to prioritize data quality over comprehensiveness.

We do not believe that power and energy measurements should include physical systems such as air conditioning, lighting, power distribution, and the like. The contributions from those systems are important, but it is much harder to draw a causal relationship or even a correlation between the power and energy consumed in physical systems and the choice of software design parameters. Also, the infrastructure to facilitate the accurate measurement of physical systems is often not in place. Furthermore, with PUE values of around 1.2 [18, 22], the value of improving software energy efficiency is much greater than the value of improving physical systems.

Last but not least, power measurements should record real power and not apparent power. In AC systems, apparent power includes the energy flow caused by capacitive and inductive circuit elements. Such cyclic power flows return to the power source at the end of each cycle, resulting in net zero energy transfer. Power flows of this type are not available for doing any work, and thus should be excluded from power-energy measurements. Real power measures only the non-zero net power transfer that is available for doing useful work.

4. METHODOLOGY

Guided by the experimental philosophy in Section 3, we designed our experiments as below.

Our cluster includes up to 13 machines, each a Sun Microsystems X2200 M22 machine, with two dual-core AMD Opteron Processor 2214 at 2.2GHz, 3.80GB RAM, 250GB SATA Drive, running Linux 2.6.26-1-xen-amd64. Machines in the cluster are connected to each other via 1Gbps Ethernet through a single switch.

We use Hadoop distribution 0.18.2 running on Java version 6.0, update 11. Hadoop 0.18.2 was the latest distribution when we began our experiments. To ensure data comparability, we decided against switching to newer distributions. For our Nutch experiments, we used Hadoop distribution 0.19.1, which comes prepackaged with Nutch. Unless otherwise noted, we used default, out of the box configuration parameters for all systems.

We run Hadoop in a controlled environment, motivated by the reasons mentioned in Section 3.3. The machines we use are a part of a shared cluster, so we could not guarantee that there were no background tasks on the machines we used, nor that there were no background network traffic. However, we perform our experiments during periods of low or no background load on the cluster, and closely monitored the CPU, disk, and network load during our experiments. When we detect any activity not due to Hadoop, we stop data collection and repeat the measurement at a later time. We run Hadoop with no virtualization.

Our Hadoop jobs includes sort, an IO-bound task that exercises all IO paths, and comparable with the existing JouleSort benchmark. We used the standard terasort format of 100 bytes records with 10 bytes keys. Also, we developed our own artificial workloads to try to isolate different parts of the IO system. We also ran Hadoop jobs that performed a configurable amount of HDFS read, HDFS write, and data shuffle. In addition, we ran Nutch, a web crawler that is representative of common workloads at leading Internet search engine companies [9].

For energy measurement, we used a Brand Electronics Model 21-1850/CI power meter. It has 1W power resolution with logging capabilities, and we set it to sample at 1Hz. It is attached to the wall socket of one of the machines in our cluster. We do not yet have enough power meters to monitor all machines in the cluster, nor can we monitor the network switch. We are making the methodological simplification that because the machines in the cluster are homogenous, we expect the power consumption of each machine to be homogenous also. We take multiple measurements. Thus, given the randomized nature of Hadoop scheduling and HDFS block placement algorithms, the variation in the performance of the machines should be captured in the experimental confidence intervals.

We did not measure detailed CPU, memory, disk, and network utilization during for workloads. Collecting such data and trying to correlate them with software parameters is the subject of future work.

Unless otherwise noted, all data points we present come from five repeated readings, and the error bars represent 95% confidence intervals.

5. RESULTS

In this section, we characterize Hadoop energy efficiency performance from several perspectives. We begin with measuring the scalability behavior of sort and the Nutch web crawler, as well as artifical workloads constructed to stress different data paths. We then examine the effect of changing static Hadoop configuration parameters, including HDFS replication and block size. We also investigate whether the input size of jobs would impact performance.

5.1 Scalable Benchmark

Our first set of experiments involves changing the number of workers only. We ran sort and Nutch. These experiments investigate whether adding more workers would improve the performance of each workload. Each worker runs MapReduce tasks and acts as a HDFS data node. We have a dedicated machine that is the master.

5.1.1 Sort

Figure 1 shows the performance of sort. Input data size is fixed at 10GB. There are several interesting features. Job duration or finishing time decreases with an increased number of workers. Total energy decreases with the number of workers, and as expected, total power increases linearly with the number of workers.

Figure 2 shows the power per node for both workers and master. The power remains near constant as we increase the number of workers in the cluster. Also, we added a data point for zero worker - measuring the average idle power of the machine. We see from the graphs that the average power when running a job is only marginally higher than the idle power. This echoes the finding in [12] that most systems today are not energy proportional.

Our results for sort invites the question whether one can reduce energy consumption simply by adding more workers. We believe that this is not the case. Our results are somewhat distorted by the fact that we have included the energy consumption of the master in our measurements. We see from Figure 2 that power consumed by the master is relatively fixed. So as we increase the number of workers, the relative power consumption of the master is ”amortized” over the workers. To illustrate this point, we re-display in Figure 3 the data from Figure 1, with the contribution of the master subtracted away.

The job duration graph remains unchanged. However, we see that the total energy is much more of a flat line rather than a pronounced downward trend. We can reason that the one worker case should consume less energy because it does no network data transfer. The total power is still a linear line, shifted downwards vertically from the line in Figure 1 by roughly 200W, i.e. the power consumption of a worker.

When the energy consumption workload is relatively constant for a constant workload sized, we can say that the workload has been designed to scale to more workers. For sort, this is the case.

In terms of energy per work unit, Hadoop sort with default configuration parameters can achieve around 100 sorted records per joule. For comparison, the current JouleSort benchmark winner can do 11k sorted records per joule. Default Hadoop is two orders of magnitude behind.

5.1.2 Nutch

Nutch is a web crawler and indexer. At its core, it performs a loop with several iterations. It begins by injecting URLs

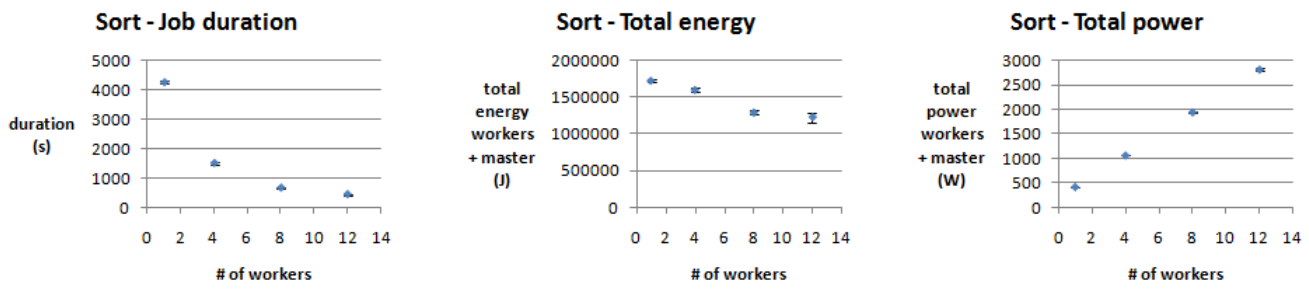

Figure 1: Sort performance

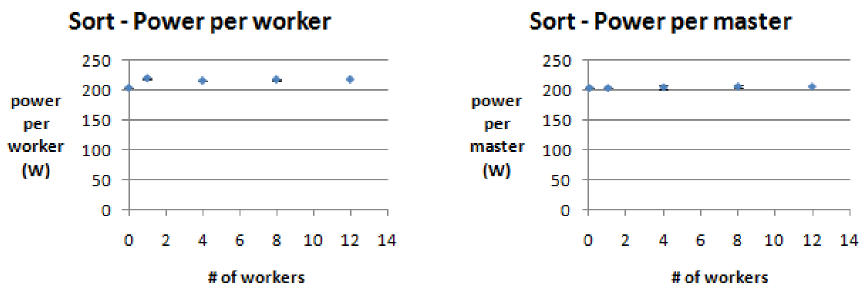

Figure 2: Sort performance - power per machine

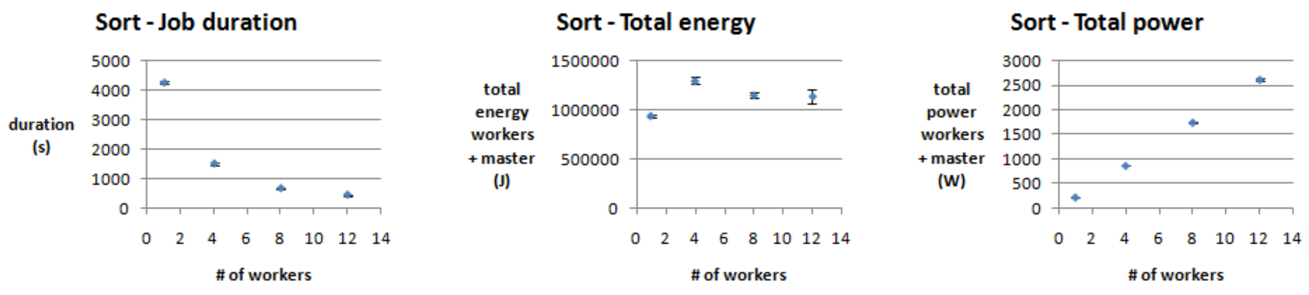

Figure 3: Sort performance - workers only

into a crawl database, in order to bootstrap it. Then, it generates a set of URLs to fetch, fetch the content at the URLs, parse the content, update the crawl database with new URLs, invert links from the content on the URLs, and finally indexes the URL content before beginning another iteration.

For our experiments, we bootstrap Nutch to crawl and index URLs anchored at www.berkeley.edu, going up to a depth of 7, and following the top 2000 links on each page. Our job duration, energy, and power measurements are in Figure 4.

These results are very surprising to us. They mean that Nutch, as we have run it, is not scalable to a large number of workers. The job still finishes, but using more workers gives no benefit in either job duration or energy or power. So we might as well run Nutch on a single node always.

It would be easy to dismiss our results either as a misconfiguration on our part, or poor design on Nutch's part. As we will show later, the results are likely because our crawl and index dataset is not large enough to take advantage of the full potential for parallelism offer by Nutch. For operators running workloads similar to Nutch but with significantly larger crawl and index dataset size, it would be very valuable to repeat our experiments and investigate whether our observations are distorted by small dataset, or they reveal design limitations in Nutch.

5.2 Isolated IO Tasks

We are unsatisfied with the somewhat "black-box" perspective that we initially took while looking at the sort and Nutch results. We would like to develop a general way in which we can isolate each part of a MapReduce/Hadoop job. The compute part of each MapReduce/Hadoop job is very job-specific and hard to generalize. However, the IO operations are common across all MapReduce/Hadoop jobs. Therefore, we devised three workloads to stress each major part of the MapReduce/Hadoop IO path:

HDFS read: Stressing the HDFS disk reads necessary before a Map task can begin.

Shuffle: Stressing the network, memory, and if necessary, the disk used for shuffle data between Map tasks and Reduce tasks.

HDFS write: Stressing the HDFS disk writes necessary after a Reduce task completes.

Each of these workloads stress one part of the IO path, and does nothing else. They are all modified from the sort and randomwriter examples that come pre-packaged with recent Hadoop distributions. The measurements for the workloads are in Figure 5. To ensure comparability with our earlier sort results, we set the jobs to read, shuffle, or write 10GB of data using terasort record format.

The most surprising result is that increasing the number of workers would not decrease the job duration of HDFS write, even though HDFS read and shuffle both run faster with more workers. HDFS write exhibits great variation in job duration, mirroed in the great variation in the energy consumed for HDFS write. HDFS write also consumes more energy at a large number of workers. For HDFS read and shuffle, the energy is relatively constant.

To more convincingly display the relative scalability behavior of the different isolation workloads, we constructed Figure 6, showing the duration and energy contributions of each of HDFS read, write, and shuffle as a fraction of the sum, e.g. for the fraction that HDFS read contributes to the duration,

$$Fraction(hdfsRead) = \frac{t_{hdfsRead}}{t_{hdfsRead} + t_{shuffle} + t_{hdfsWrite}}$$

where t_{TASK} is the duration of $TASK$.

In Figure 6, the job duration and energy fractions track each other. If all three jobs scale in a similar fashion, the relative fractions should be the fixed as we increase the number of senders. We see that indeed HDFS write contributes a larger and larger fraction of the duration and energy relative to the total sum. In other words, HDFS write does not scale to more workers at the same rate as HDFS read and shuffle.

We demonstrate in Section 5.3 that the poor scalability and great variation in the HDFS write performance is removed when we reduce HDFS replication level.

5.3 HDFS Replication

We want to investigate whether decreasing HDFS replication brings about a clear benefit. in terms of time efficiency and energy efficiency. We repeat our sort and isolation jobs using the same input size and configuration parameters, changing only the HDFS replication level. We run the experiments for 12 workers and a fixed data size of 10GB. Our results are in Figure 7.

The total power remains identical across different HDFS replication levels for all jobs, a result that we expected. Thus, any energy savings would come entirely from decreases in job duration. Indeed, there is a strong correlation in the shape of the duration and energy graphs. In Section 6.4, we have a more extensive discussion regarding the tight correlation between energy and job duration.

The measurements show that decreasing HDFS replication level brings negligible benefit for HDFS read and shuffle, but a clear benefit for HDFS write both in terms of the average performance and the variability in performance. We can explain this observation by the way HDFS replicate blocks. When a block is written to HDFS, the machine doing the local write begins writing to disk immediately, and tells a different HDFS data node to begin writing the second replica. The second machine then begins writing, and tells a third machine to begin writing the third replica. The chain continues until HDFS has reproduced the required number of replicas. When each of the replication data nodes finishes writing, it notifies the data note that initially requested the replicas. The HDFS write call would return only when all replica data nodes reported completion. When we decrease the replication level, the chain is "shorter", and both the average and variation in finishing time would decrease. Thus,

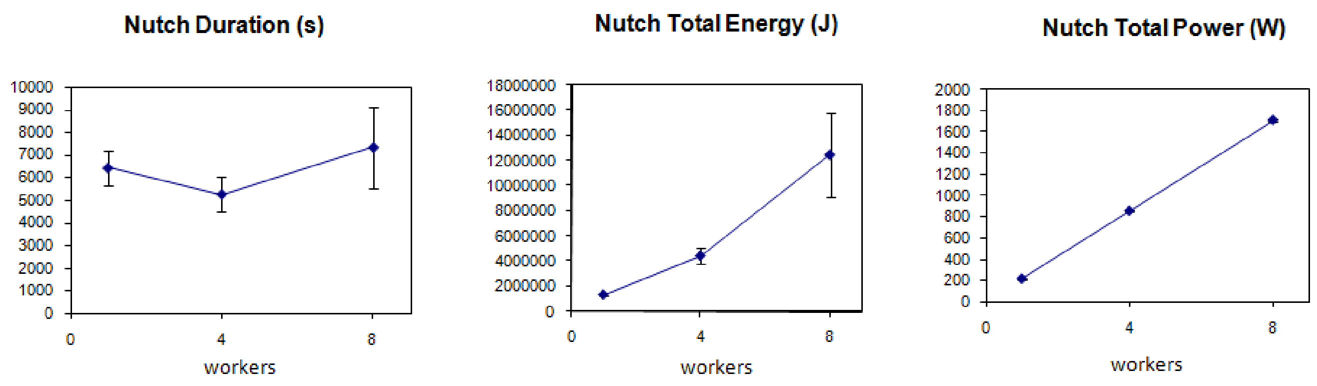

Figure 4: Nutch performance - workers only

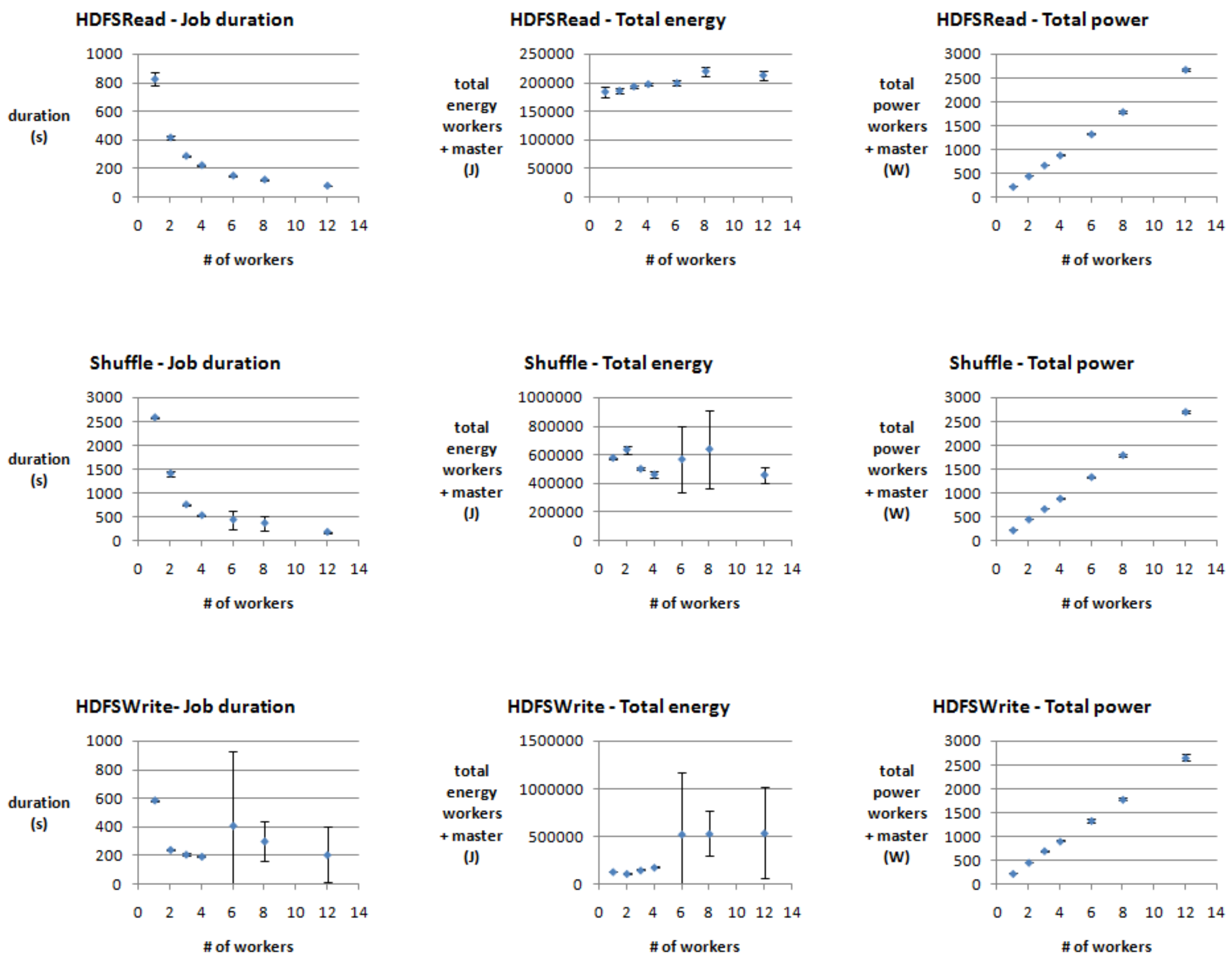

Figure 5: Isolation jobs performance - workers only

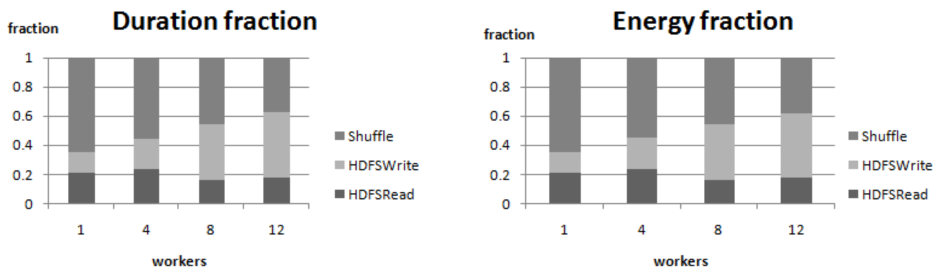

Figure 6: Isolation jobs performance as relative fractions of sum of the jobs

we believe the HDFS replication mechanism is responsible for the poor scalability for HDFS write, as shown by the data in Section 5.2. Later, we will present in Section 6.3 a more detailed quantitative discussion about how to identify the optimal HDFS replication level.

The data point for 4-fold HDFS replication is off the trend for HDFS write and HDFS read. HDFS replication level above 3 is usually not used, because it imposes additional storage space overhead for negligible additional benefit in storage failure tolerance. We are yet to pin point why 4-fold HDFS replication exhibit behavior different from the other replication levels.

In Figures 8 and 9, we reproduce the duration and energy fraction graph using data for HDFS replication of 2 and 1, respectively. Recall that in Figure 6, HDFS write is the dominant fraction at a large number of workers. In Figures 8 and 9, shuffle is the dominant fraction and the scaling bottleneck. This means that if a HDFS replication lower than 3 is acceptable, then the greatest opportunity for further improvement would be in the shuffle part of the IO path.

5.4 HDFS Block Size

One other HDFS parameter that can be easily changed is HDFS block size. Different HDFS block sizes will also have an impact on the performance of various workloads. We repeated our experiments for 12 workers, 10GB input data size, default 3-fold HDFS replication, and varied the HDFS block size from 16MB to 1GB. Our results are in Figure 10.

The measurements indicate that large HDFS block size brings a clear benefit for HDFS read and write workloads, but negligible benefit for shuffle. Thus, for jobs that have a large amount of HDFS input/output data, such as sort, the HDFS block size should increase appropriately with the workload size.

5.5 Input Size

Our observations with regard to Nutch in Section 5.1.2 lead us to suspect that the size of a workload may impact energy efficiency. The inescapable overhead in MapReduce/Hadoop would be amortized across large jobs, and prevent small jobs from scaling. To investigate the trade off associated with workload size, we repeat the sort workload for input data sizes of 20GB, 5GB, 1GB, 500MB, and 100MB. To ensure comparability with our earlier sort results for 10GB, we run Hadoop with default parameters, in particular with a HDFS replication level of three replicas. The results are in Figure 11. We omit the power results because we expect the power to be independent of workload size, and the experimental measurements confirm this expectation.

The data is somewhat hard to read, because the input size varies across two orders of magnitude, and the duration and energy is expected to also vary across the same scale. To make the results more readable, we normalize the duration and energy, such that the duration and energy for each data point is expressed as a fraction of the maximum duration and energy for that particular data input size. This gives us a series of lines for each data input size, with the maximum of the line bounded by 1, and allowing us to more easily identify any trends in the data. The normalized graphs are in Figure 12.

In the normalized energy curves, we can see the effect of inherent MapReduce/Hadoop overhead. The lines for 20GB and 10GB are generally constant - the increase in parallelism overrides the effect of inherent MapReduce/Hadoop overhead. The line for 5GB is constant up to 4 workers, then increasing thereafter - increase in parallelism dominates until the work done on each worker becomes comparable and then negligible relative to the overhead. After this point the energy consumed increases. The 1GB and 500MB lines display similar behavior. For 100MB, the overhead dominates, i.e., the job takes nearly the same amount of time to finish regardless of the number of workers, and the energy consumed increases steadily.

Figure 13 shows sorted records per joule for the different input sizes. There are several noteworthy points. For small input sizes, we can increase the records sorted per joule by reducing the number of workers. This suggests that the overhead of coordinating workers and distributing data to the workers is a bottleneck in the system. For larger input sizes, the records sorted per joule is constant relative to the number of workers. This suggests that the overhead associated with many workers is balanced out by increased parallelism for large input sizes. However, the records sorted per joule for large input sizes is still less than that for small input size and few workers. This suggests that the data locality associated with fewer workers is more important than the increased parallelism associated with many workers. Lastly, the curve for 100MB is significantly lower than the rest, suggesting workloads that involve many small jobs would significantly benefit from decreased overhead.

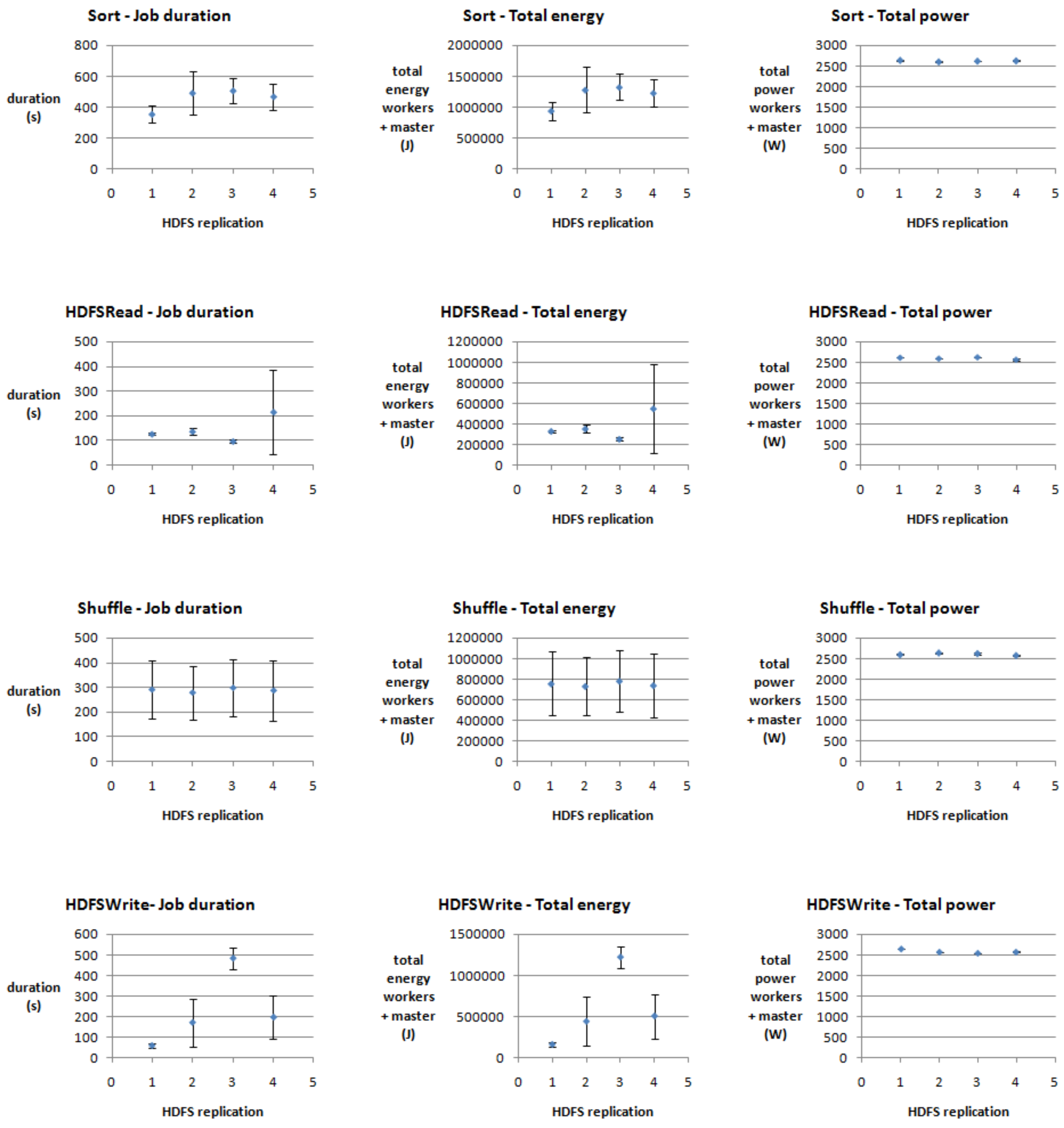

Figure 7: Sort and isolation jobs performance for different HDFS replication - workers only

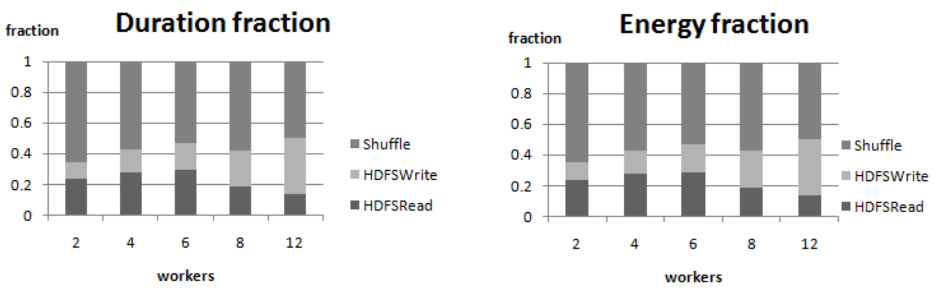

Figure 8: Isolation jobs as relative fractions of sum of the jobs - HDFS replication of 2

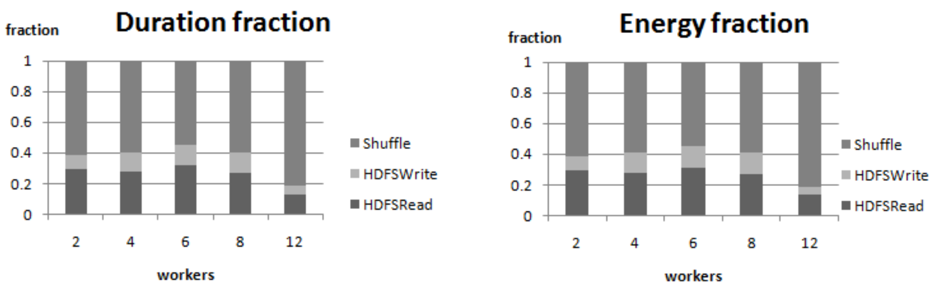

Figure 9: Isolation jobs as relative fractions of sum of the jobs - HDFS replication of 1

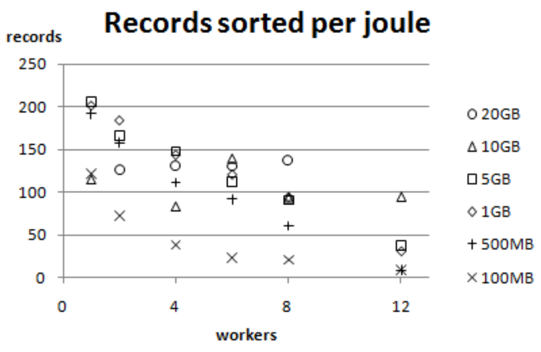

Figure 13: Records sorted per joule for different input data size - workers only

5.6 Slow Nodes

We made an accidental discovery when monitoring HDFS block placement. One node in our cluster persistently gets a smaller number of assigned blocks. Since HDFS block placement is supposed to be random, it would mean that the node reports less frequently that it is ready to receive new blocks. To quantify this difference in block placement, we run the HDFS write job for 10GB, 10 workers, 20 repeated measurements. The average number of blocks written to each node is in the left graph in Figure 14, with the slow node highlighted. We normalize the block numbers such that the amount of variation does not depend on the data input size. We see that node r32 is much slower, and the high-end of its 95% confidence interval does not overlap with the low-end of the 95% confidence interval for any other nodes.

We remove node r32 from the cluster, and repeat the measurements. The block allocations are now more even, as evidenced from the right graph in Figure 14. Table 1 shows the job duration, energy, and power comparison before and after the slow node is removed. All metrics show improvements when the slow node is removed. Thus, it is important to identify slow nodes in the cluster and deal with them appropriately. We can get a performance improvement simply by removing slow nodes.

6. QUANTITATIVE MODELS

In this section, we provide quantitative models to answer several questions critical to the efficient operation of a Hadoop cluster. We show that we can predict the scalability trends in the IO performance of a job from the amount of its input read, output write, and intermediate shuffle data. We also demonstrate that we can provision cluster size in a scientific manner based on expected workload characterstics. We further provide quantitative guidance on optimizing the level of HDFS replication without compromising reliability. Lastly, we explore the relationship between time efficiency and energy efficiency, and argue that under simplified conditions, the two are equivalent.

6.1 Predicting Energy Consumption

From our results, we noticed that a good way to predict the performance of a sort job is to add the performance of the HDFS read, shuffle, and HDFS write jobs of the same

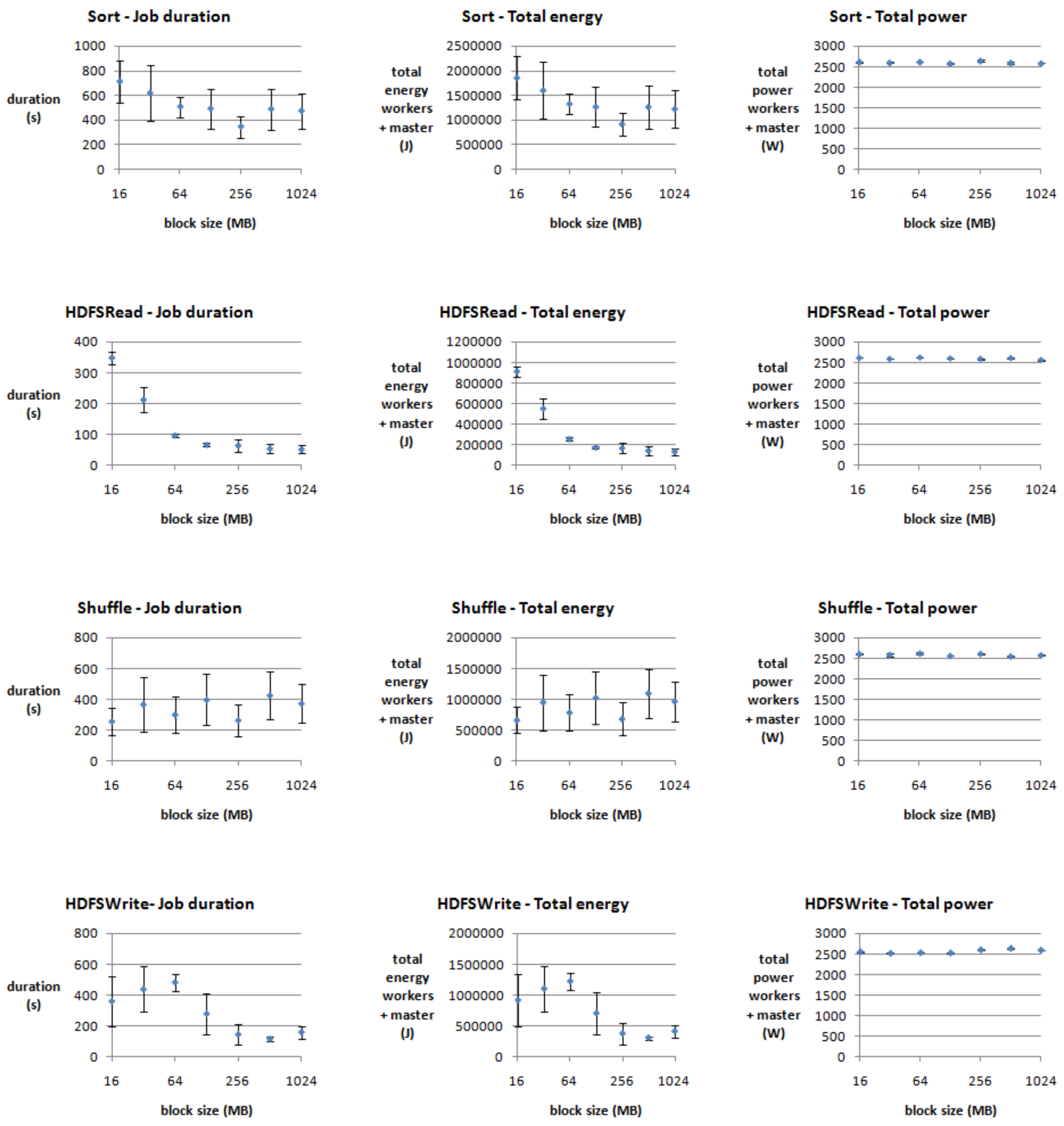

Figure 10: Sort and isolation jobs performance for different HDFS block size - workers only

Table 1: Performance improvement of removing a slow node

Experiment	Duration (s)	95% confidence interval	Energy (J)	95% confidence interval	Records / J	Power (W)	95% confidence interval
With r32	388	73.7	827039	157170	129.8	2134	1.8
Without r32	301	81.2	648813	174540	165.4	1941	5.6

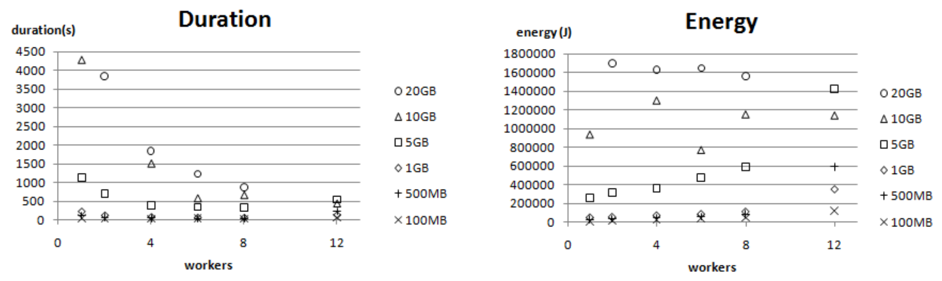

Figure 11: Sort performance with different data input size - workers only

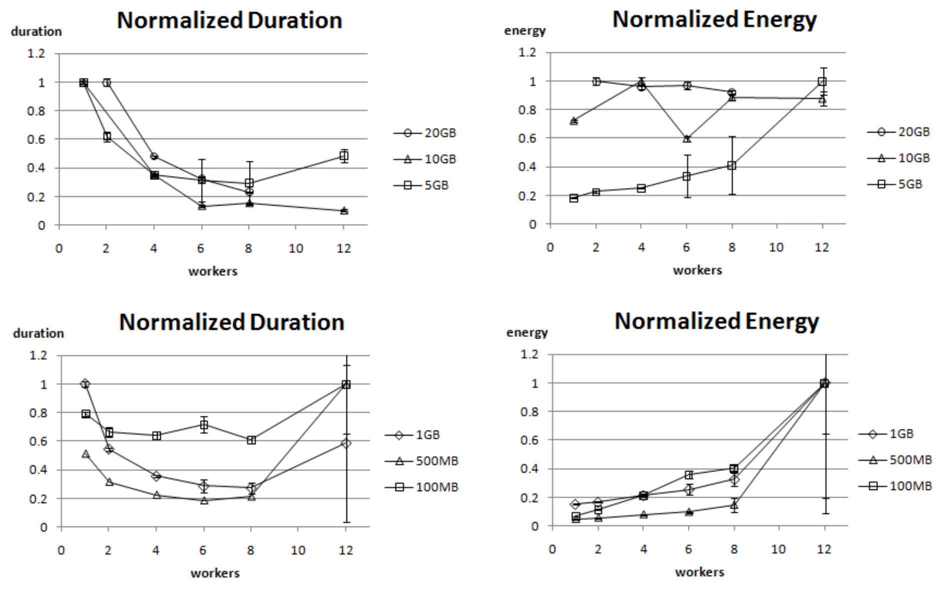

Figure 12: Normalized sort performance with different data input size - workers only

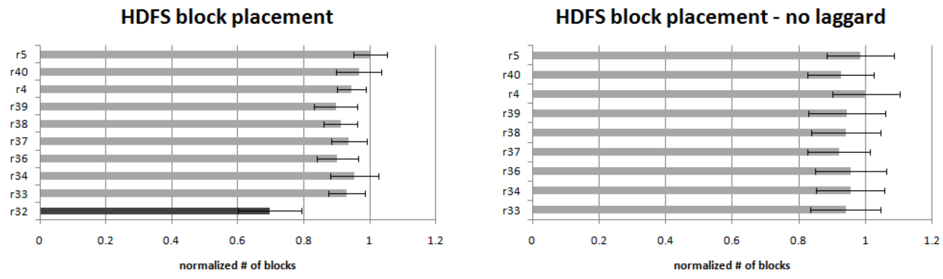

Figure 14: HDFS block placement for HDFS write with slow node (left) and with slow node removed (right)

workload and cluster size. That is,

$$\begin{aligned}
t_{sort} &\approx t_{hdfsRead} + t_{shuffle} + t_{hdfsWrite} \\
E_{sort} &\approx E_{hdfsRead} + E_{shuffle} + E_{hdfsWrite} \\
P_{sort} &\approx average[P_{hdfsRead}, P_{shuffle}, P_{hdfsWrite}]
\end{aligned}$$

where t, E, P are respectively time, energy, and power for each part of the datapath, and the average for power is a weighted average with the weights determined by the relative durations associated with each part of the data path. We applied this model to data for HDFS replication levels of 3, 2, and 1, and the results are in Figures 15, 16, 17.

This simple model is not exact, and in some cases there are significant differences between the predicted and measured performance. In particular, the model assumes no overlap in the execution of different IO tasks. Despite this, the model is effective at predicting trends. The reason is that sort does no computation at all, and all the work done in a sort job is exactly the sum of HDFS read, shuffle, and HDFS write. Over-predictions exist because IO tasks are overlapping, unaccounted in the model. Under-predictions exist because there are coordination overhead in MapReduce, also unaccounted in the model. The overall trend, however, is reflected in the predictions.

For MapReduce/Hadoop jobs in general, if we know the amount of input, shuffle, and output data, we should be able to use this method to predict the duration, energy, and power trends associated with the IO parts of a MapReduce/Hadoop job, as below.

$$\begin{aligned}
t_{IO} &\approx t_{hdfsRead} + t_{shuffle} + t_{hdfsWrite} \\
E_{IO} &\approx E_{hdfsRead} + E_{shuffle} + E_{hdfsWrite} \\
P_{IO} &\approx \text{average}[P_{hdfsRead}, P_{shuffle}, P_{hdfsWrite}]
\end{aligned}$$

For MapReduce tasks that have straightforward map and reduce functions where the amount of computation is proportional to the amount of data, the sums and average above would include an additional term representing the non-IO compute part of MapReduce jobs.

6.2 Provisioning Machines

A question that frequently faces Hadoop operators is how many machines to allocate to a cluster. Alternatively, for the Hadoop tasks scheduler, the question is how many machines to allocate to a particular job. Our measurements suggested a way to answer this question. We illustrate the method with a simplified example.

Suppose we have a cluster dedicated to running sort. The sort jobs have relatively constant sizes, and the jobs arrive at random times with a known average interval. Such job size and arrival characteristics are representative of analytic workloads that process data of regular sizes at regular intervals. The question to answer is how many machines to assign to the MapReduce/Hadoop cluster.

Job duration would be an irrelevant metric, unless there are constraints on workload response times. Power consumption would also be secondary, unless there are power constraints based on cooling limits and power provision ceilings. The primary metric in our example should be energy per job. We do not turn off the system when it is idle, because the jobs arrive at random times, even though the average rate of arrival is known. Thus, the energy consumed per job should include the energy consumed while the system is idle. We define the following variables and functions:

$$\begin{aligned}
N &= \text{number of workers} \\
D(N) &= \text{job duration, a function of } N \\
P_a(N) &= \text{power when active, a function of } N \\
P_i &= \text{power when idle, should be near constant} \\
T &= \text{average job arrival interval} \\
E(N) &= \text{energy consumed per job, a function of } N
\end{aligned}$$

Using these functions and variables, we can express the expected energy per job as

$$E(N) = D(N)P_a(N) + (T - D(N))P_i$$

This is a straight forward linear sum multiplying job duration by active power and idle duration by idle power. We optimize $E(N)$ for the range of N such that $D(N) \leq T$.

To give a numerical example using actual data, suppose the sort jobs are 10GB terasort format, and T is 1000 seconds. Taking the data points in Figure 1, we optimize for the choice of $N = 8$ or $N = 12$. The data for 8 and 12 workers are

$$\begin{aligned}
D(8) &= 661 \text{ s} \pm 15 \text{ s (95\% confidence)} \\
D(12) &= 436 \text{ s} \pm 25 \text{ s 95\% confidence)} \\
Pa(8) &= 1739 \text{ W} \pm 5 \text{ W 95\% confidence)} \\
Pa(12) &= 2611 \text{ W} \pm 18 \text{ W (95\% confidence)} \\
Pi &= 203 \text{ W} \pm 1 \text{ W (95\% confidence, per worker)}
\end{aligned}$$

Substituting, we get

$$\begin{aligned}
E(8) &= (661 \text{ s})(1739 \text{ W}) + (1000 \text{ s} - 661 \text{ s})(203 \text{ W}) \\
&= 1218 \text{ kJ per job} \pm 33 \text{ kJ} \\
&= 0.338 \text{ kWHr per job} \pm 0.009 \text{ kWHr} \\
E(12) &= (436 \text{ s})(2611 \text{ W}) + (1000 \text{ s} - 436 \text{ s})(203 \text{ W}) \\
&= 1252 \text{ kJ per job} \pm 76 \text{ kJ} \\
&= 0.348 \text{ kWHr per job } \pm 0.021 \text{ kWHr}
\end{aligned}$$

Since $E(8) < E(12)$, we should provision 8 machines for this particular job stream, even though it takes longer to finish any single job.

We can quantify the potential savings of such optimizations. Suppose that we lack the information necessary to do the above calculation. Thus, we provision 8 machines half the time and 12 machines half the time. This would mean an average 0.05 kWHr saved for each job. For a job arrival rate of $T = 1000$s per job, this translates to 1577 kWHr saved per year. For an average industrial electricity retail price of 7.01 centers per kWHr in 2008 [1], this means approximately annual savings of $110 per machine using well-informed, straight-forward provisioning decisions.

The analysis process we illustrate here is extensible to workloads with different types of jobs, different size of jobs, and different arrival model for jobs. If the level of energy and financial savings here are representative, then for datacenters of order 100,000 servers, the aggregated savings are on the order of $11 million yearly.

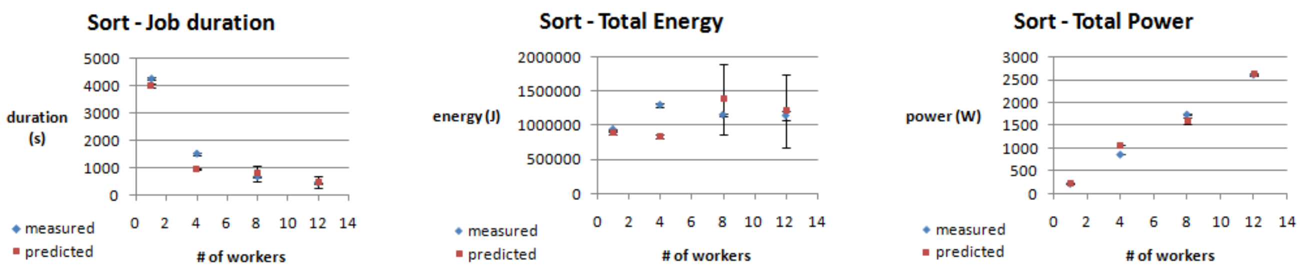

Figure 15: Predicting sort performance - HDFS replication 3

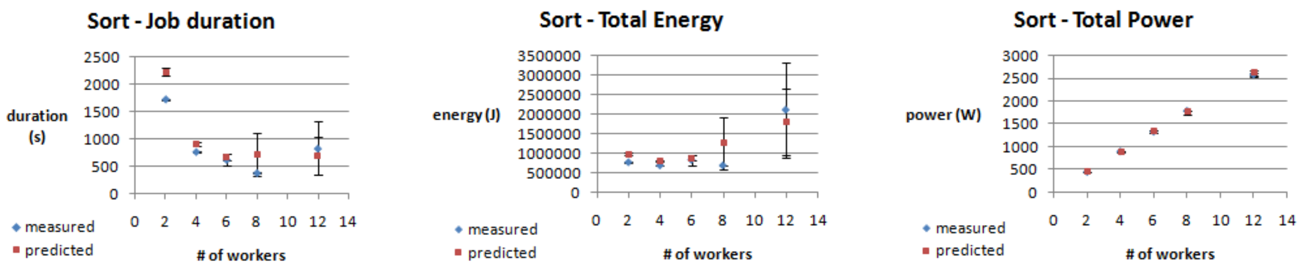

Figure 16: Predicting sort performance - HDFS replication 2

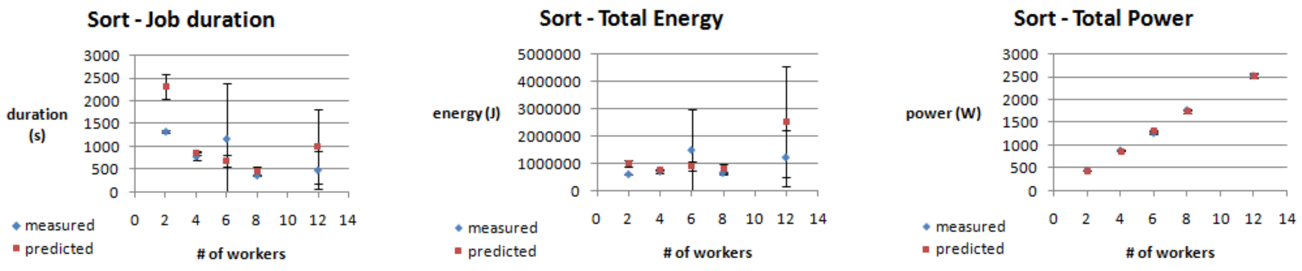

Figure 17: Predicting sort performance - HDFS replication 1

6.3 Replication and Reliability

In light of the results in Section 5.3, we will provide some methods to systematically optimize the HDFS replication level. For production Hadoop deployments, optimization of this type should be motivated by historical machine failure characteristics. In the absence of such published data, we will focus on trends and trade-offs. To ensure the generality of our analysis, we will consider two scenarios.

First, we will look at Hadoop deployments on top of a reliable, archival storage system. HDFS serves as the workspace of computation pipelines, and the archival storage system serves as the permanent repository for input and output data. Such deployments would be required if the Hadoop clusters are transient, but the data needs to be always available. These deployments would also make sense for running Hadoop on bulk archival data, for which HDFS replication may be impossible due to storage capacity. Our lab uses a couple of Hadoop deployments of this type. On our local cluster, the archival storage is a networked drive backed by ZFS. Typically we copy input data to HDFS at the start of a computation process, and copy output data to ZFS at the end. We also have a deployment on Amazon's EC2 platform, where the archival storage is S3, and the Hadoop cluster runs on EC2 instances. We can either take S3 data and put it in HDFS, or use S3 directly in place of HDFS. In both deployments, we need the separate storage because the Hadoop cluster is transient. The optimization focus for these deployments is performance, since we can always recover from machine failures by re-copying the data from the separate reliable storage and repeating the computation.

Second, we will look at Hadoop deployments that have HDFS as its only storage system. These deployments typically represent persistent Hadoop clusters, and any machine failures that gets past the HDFS replication mechanisms will result in data loss. The optimization focus for these deployments is preventing data loss without excessively harming performance.

Our analysis indicate that for Hadoop deployments on top of a reliable storage system, we should *always* run at low HDFS replication, even when the cost of staging data in and out of HDFS is considered. However, for Hadoop deployments that have HDFS as its only storage system, we should not lower the HDFS replication level because doing so would significantly compromise data durability.

6.3.1 HDFS with separate reliable storage

We focus on a computation process with three stages - copy to HDFS from the reliable storage at the beginning, some computation tasks on Hadoop, and write to archival storage at the end. For simplicity, when machines in our Hadoop cluster fail and results in data loss in HDFS, we recover by restarting the computation process with input copied from the separate reliable storage. This recovery model would not apply if we use Hadoop to generate non-reproducible data such as logs, user interaction streams, etc.

In this scenario, we alway want to run at lower HDFS replication, provided that we can use this recovery process to regenerate the lost data. The result is intuitive because the separate reliable storage makes it redundant for HDFS to achieve data durability through replication.

To derive this insight analytically, we introduce the following variables:

$$\begin{aligned}
T_{3fold} &= \text{normal process duration with 3-fold replication}\\
T_{lower} &= \text{normal process duration with lowered replication}\\
T'_{3fold} &= \text{recovery time with 3-fold replication}\\
&= 2T_{3fold} \text{ max for the recovery process described}\\
T'_{lower} &= \text{recovery time with lowered replication}\\
&= 2T_{lower} \text{ max for the recovery process described}\\
E_{3fold} &= \text{normal energy consumed with 3-fold replication}\\
E_{lower} &= \text{normal energy consumed with lowered replication}\\
E'_{3fold} &= \text{recovery energy with 3-fold replication}\\
&= 2E_{3fold} \text{ max for the recovery process described}\\
E'_{lower} &= \text{recovery energy with lowered replication}\\
&= 2E_{lower} \text{ max for the recovery process described}\\
f_{3fold} &= \text{failure probability with 3-fold replication}\\
f_{lower} &= \text{failure probability with lowered replication}
\end{aligned}$$

The upper bounds for the recovery duration and energy is twice the normal duration and energy because for the scenario we described, recovery from failure is analogous to executing the job twice. Using these variables, the expected process durations are

$$\begin{aligned}
T_{expected\ 3fold} &= (1-f_{3fold})T_{3fold} + f_{3fold}T'_{3fold}\\
&= (1-f_{3fold})T_{3fold} + 2f_{3fold}T_{3fold}\\
&= (1+f_{3fold})T_{3fold}\\
T_{expected\ lower} &= (1-f_{lower})T_{lower} + f_{lower}T'_{lower}\\
&= (1-f_{lower})T_{lower} + 2f_{lower}T_{lower}\\
&= (1+f_{lower})T_{lower}
\end{aligned}$$

Likewise, the expected energy consumption is

$$\begin{aligned}
E_{expected\ 3fold} &= (1-f_{3fold})E_{3fold} + f_{3fold}E'_{3fold}\\
&= (1-f_{3fold})E_{3fold} + 2f_{3fold}E_{3fold}\\
&= (1+f_{3fold})E_{3fold}\\
E_{expected\ lower} &= (1-f_{lower})E_{lower} + f_{lower}E'_{lower}\\
&= (1-f_{lower})E_{lower} + 2f_{lower}E_{lower}\\
&= (1+f_{lower})E_{lower}
\end{aligned}$$

The apparent performance-reliability tradeoff is that $f_{3fold} < f_{lower}$, but $T_{3fold} > T_{lower}$, and $E_{3fold} > E_{lower}$. For most systems, failure probabilities are low. This means if we approximate

$$\begin{aligned}
(1+f_{3fold}) &\approx 1\\
(1+f_{lower}) &\approx 1
\end{aligned}$$

then

$$\begin{aligned}
T_{expected\ 3fold} &\approx T_{3fold}\\
T_{expected\ lower} &\approx T_{lower}\\
E_{expected\ 3fold} &\approx E_{3fold}\\
E_{expected\ lower} &\approx E_{lower}
\end{aligned}$$

In other words, if failure probabilities are small, $T_{\mathit{3fold}} > T_{\mathit{lower}}$ and $E_{\mathit{3fold}} > E_{\mathit{lower}}$, then we should *always* run at lowered HDFS replication.

The cost of staging data in and out of HDFS has been implicitly accounted in the analysis above. We can rewrite the process duration and energy consumed as

$$\begin{aligned}T_{\mathit{3fold}} &= T_{\mathit{3fold,\ Hadoop}} + T_{\mathit{3fold,\ data\ staging}}\\ T_{\mathit{lower}} &= T_{\mathit{lower,\ Hadoop}} + T_{\mathit{lower,\ data\ staging}}\\ E_{\mathit{3fold}} &= E_{\mathit{3fold,\ Hadoop}} + E_{\mathit{3fold,\ data\ staging}}\\ E_{\mathit{lower}} &= E_{\mathit{lower,\ Hadoop}} + E_{\mathit{lower,\ data\ staging}}\end{aligned}$$

Thus, we can expand the final approximations as

$$\begin{aligned}T_{\mathit{expected\ 3fold}} &\approx T_{\mathit{3fold,\ Hadoop}} + T_{\mathit{3fold,\ data\ staging}}\\ T_{\mathit{expected\ lower}} &\approx T_{\mathit{lower,\ Hadoop}} + T_{\mathit{lower,\ data\ staging}}\\ E_{\mathit{expected\ 3fold}} &\approx E_{\mathit{3fold,\ Hadoop}} + E_{\mathit{3fold,\ data\ staging}}\\ E_{\mathit{expected\ lower}} &\approx E_{\mathit{lower,\ Hadoop}} + E_{\mathit{lower,\ data\ staging}}\end{aligned}$$

Data staging involves two parts - reading from and writing to the separate storage system, which incurs a fixed cost regardless of HDFS replication level, and writing to and reading from HDFS, which incurs a lower or equal cost for lower HDFS replication levels, as indicated in Section 5.3. Hadoop jobs also involve two parts - computation of the map and reduce functions, which incurs a fixed cost regardless of HDFS replication level, and IO, which incurs a lower or equal cost for lower HDFS replication levels, also as indicated in Section 5.3. Thus we expect $T_{\mathit{3fold}} > T_{\mathit{lower}}$ and $E_{\mathit{3fold}} > E_{\mathit{lower}}$, and our earlier conclusion already accounts for the cost of staging data in and out of HDFS.

To quantify the size of potential savings, let us consider a worst case example with large cluster size, high failure probability, 1-fold replication, and marginal increase in performance from reduced replication level. We can express f_{lower} as

$$f_{\mathit{lower}} = 1 - (1-p)^N$$

where p is the probability of any single machine failing during a job, and N is the size of the cluster. This expression assumes that machine failures are indepdent. If they are highly dependent, then it does not make sense to have high replication - the first failure will trigger failures of replicas. Let us assume $N = 1000$ for a large cluster, $p = 0.0001$, i.e. on average a machine fails every 10,000 jobs. For simplicity let us also assume $T_{\mathit{lower}} = 0.85T_{\mathit{3fold}}$ and $E_{\mathit{lower}} = 0.85E_{\mathit{3fold}}$, i.e. 15% improvement in duration and energy from lowered HDFS replication. Substituting, we get

$$f_{\mathit{lower}} = 1 - (1-0.0001)^{1000} = 0.095 = 9.5\%$$

i.e. we expect to repeat every 10th job because some machine has failed. Substituting again, we get

$$\begin{aligned}T_{\mathit{expected\ lower}} &= (1 + f_{\mathit{lower}})T_{\mathit{lower}}\\ &= (1 + 0.095)0.85T_{\mathit{3fold}}\\ &= 0.93T_{\mathit{3fold}}\\ E_{\mathit{expected\ lower}} &= (1 + f_{\mathit{lower}})E_{\mathit{lower}}\\ &= (1 + 0.095)0.85E_{\mathit{3fold}}\\ &= 0.93E_{\mathit{3fold}}\end{aligned}$$

Thus, even in this artificial example when we assume failure characteristics to heavily favor increased HDFS replication, we still get a 7% benefit from operating at a lower replication. Note that there is a 7% benefit in increased value from reduced job duration, *and* a 7% benefit in decreased cost from reduced energy consumption. For a order 100kW datacenter and a retail price of electricity of 7.01 cents per kWHr in 2008 [1], 7% savings are on the order of $5 million yearly.

Our data from Section 5.3 shows that operating at 1-fold HDFS replication leads to $T_{\mathit{lower}} \approx 0.13T_{\mathit{3fold}}$ and $E_{\mathit{lower}} \approx 0.13E_{\mathit{3fold}}$ for no penalty in HDFS read and shuffle. So we expect the benefits in real deployments to be much larger than 7%.

Also, real machines are likely to have mean time to failure longer than 10,000 jobs. When cluster size increases, the approximations involving f_{3fold} and f_{lower} become less exact, but not to an extent that would affect our conclusions. For example, if $p = 0.00001$, i.e. on average a machine fails every 100,000 jobs, then for $N = 1000$ representing a cluster of 1000 machines, we get

$$f_{\mathit{lower}} = 1 - (1-0.00001)^{1000} = 0.001 = 0.1\%$$

For $N = 10,000$ representing a cluster ten times as large, we get

$$f_{\mathit{lower}} = 1 - (1-0.00001)^{10000} = 0.01 = 1\%$$

The approximations

$$\begin{aligned}T_{\mathit{expected\ lower}} &\approx T_{\mathit{lower}}\\ E_{\mathit{expected\ lower}} &\approx E_{\mathit{lower}}\end{aligned}$$

are less exact but still valid.

In short, when failure probabilities are low, HDFS is deployed in conjunction with a separate reliable storage system, then it *always* makes sense to operate at 1-fold HDFS replication, and recover from HDFS data losses by repeating the computation.

6.3.2 HDFS as primary storage

For Hadoop deployments that have HDFS as its only storage system, any machine failures that gets past the HDFS replication mechanisms will result in data loss. Thus, data durability has priority over performance, and any improvements in performance should not come at the cost of increase risk of data loss. It turns out that when failure probabilities are constant, we need to have orders of magnitude decreases in cluster size such that 2-fold replication has the same fault resilience as 3-fold replication. For most deployments, making this change would not be worth the effort.

To illustrate, we use the variables from Section 6.3.1. Again, we assume that machine failures are independent. Otherwise, the benefit of replication is reduced, since one failure would be correlated with another.

We can express f_{3fold} as

$$\begin{aligned} f_{3fold} &= 1-(1-p)^N \\ &\quad - \binom{N}{1} p(1-p)^{N-1} \\ &\quad - \binom{N}{2} p^2(1-p)^{N-2} \\ &= 1-(1-p)^N \\ &\quad - Np(1-p)^{N-1} \\ &\quad - \frac{N(N-1)}{2} p^2(1-p)^{N-2} \end{aligned}$$

where $\binom{N}{n}$ are binomial coefficients. Each term subtracted correspond to the probability of no losses, 1 loss, and 2 losses. Similarly, we can express f_{2fold} as

$$\begin{aligned} f_{2fold} &= 1-(1-p)^N - \binom{N}{1} p(1-p)^{N-1} \\ &= 1-(1-p)^N - Np(1-p)^{N-1} \end{aligned}$$

In these expressions, the terms $(1-p)^{N-n}$ are the dominant terms while p is fixed, unless we change to different hardware. Thus, we can make $f_{2fold} \approx f_{3fold}$ if we decrease N, the size of the cluster. This makes intuitive sense because the smaller the cluster, the smaller the probability of simultaneous failures.

It turns out that with p fixed, the probability of a single loss is $O(N)$ larger than the probability of no losses, and the probability of a double loss is $O(N)$ larger than the probability of a single loss. So for large clusters, we need a huge decrease in N for 2-fold replication to have the same failure resilience as 3-fold replication.

To use a numerical example, let us assume $p = 1 \times 10^{-6}$, i.e. one in a million chance of failure on a particular job, and $N = 1000$ representing a cluster with 1000 nodes. We find $N' < N$ for which $f_{2fold} \approx f_{3fold}$. For these values, $f_{3fold} \approx 1.66 \times 10^{-10}$ numerically. Equating this to f_{2fold} and solving for N', we find that at $N' = 19$, $f_{2fold} \approx 1.71 \times 10^{-10}$. Thus, when failure probabilities are constant, we need decrease cluster size from 1000 to 19 such that 2-fold replication has the same fault resilience as 3-fold replication.

A 1000 node cluster and a 19 node cluster represent two completely different operation modes. To achieve the same computational throughput, we would have to divide the 1000 node cluster into some 50 sub-clusters, and have each of them operate jobs with no data dependencies with jobs in other sub-clusters. This is often difficult to achieve. Even if it is achieved, the job duration would be orders of magnitude longer, and the coordination overhead would likely mitigate the marginal performance gains of using a lower HDFS replication.

In short, when HDFS is the primary storage system, and data durability is important, the focus of improvement should be on doing 3-fold replication more efficiently instead of using a lower HDFS replication.

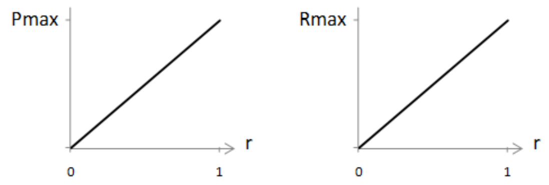

Figure 18: Power and work rate characteristics for energy proportional systems

6.4 Time Efficiency and Energy Efficiency

In various forms, we have alluded to the strong interplay between job duration, power, and energy. Since work rate is correlated with power, job duration is the inverse of work rate, and energy is the product of power and duration, one wonders whether reducing job duration is equivalent to reducing the energy consumed. We answer this question using some simple quantitative models.

We introduce the following variables and functions:

$$\begin{aligned} r &= \text{resources used in a system, ranging from 0 to 1} \\ R(r) &= \text{work rate of the system, ranging from 0 to } R_{max} \\ P(r) &= \text{power of the system, ranging from 0 to } P_{max} \\ r_l, r_u &= \text{lower and upper bounds of the operating region} \\ W &= \text{workload size} \\ E(r) &= \text{energy consumed at } r \end{aligned}$$

A fully idle system has $r = 0$ while a fully utilized system has $r = 1$. Note that usually $r_l > 0$, and $r_u < 1$. This reflects the fact that the system would always have some book-keeping tasks to do, and it is hard to drive the system to the full utilization of 1. In the operating region for our experiments, we observed P_{max} to be approximately 250W.

For ideal, energy proportional systems, $R(r)$ and $P(r)$ are as below and in Figure 18.

$$\begin{aligned} R(r) &= rR_{max} \\ P(r) &= rP_{max} \end{aligned}$$

So, for fixed workload sizes, the energy consumed is

$$E(r) = P(r)\frac{W}{R(r)} = rP_{max}\frac{W}{rR_{max}} = \frac{W}{R_{max}}P_{max}$$

That is, the energy consumed is independent of the level of utilization, and we can decrease job duration without additional energy costs. Thus, we should run our system at utilization level r_u, the upper bound of the operating region.

Realistic systems are not power proportional. They are more appropriately described by

$$\begin{aligned} R(r) &= rR_{max} \\ P(r) &= P_{idle} + r(P_{max} - P_{idle}) \end{aligned}$$

where P_{idle} is the idle power of the system, observed to be approximately 200W in our experiments. In comparison, P_{max} is approximately 250W. Graphically speaking, the behavior is as in Figure 19.

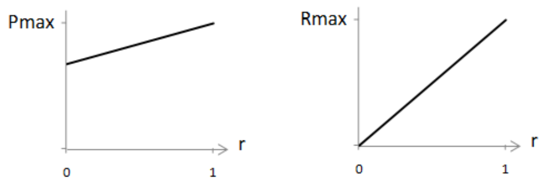

Figure 19: Power and work rate characteristics for systems that are not energy proprotional

For such a system, the energy consumed at r is

$$E(r) = P(r)\frac{W}{R(r)} = \frac{W(P_{idle} + r(P_{max} - P_{idle}))}{rR_{max}}$$
$$= \frac{W}{R_{max}}\frac{P_{idle}}{r} + \frac{W}{R_{max}}(P_{max} - P_{idle})$$

This is a decreasing function in r. So again, we should operate at r_u, the upper bound of the operating region.

The key result from our analysis is that if work rate is proportional to r, we should always run our workloads as fast as possible. We believe this analytical result to be *highly* significant, because it means that efforts to reduce energy consumption are equivalent to efforts to optimize for traditional performance metrics such as job duration. The prior work in system optimization *does not* need to be re-invented in light of energy efficiency. Our results are applicable to all real life systems that match the quantitative models we have described.

An interesting side result is that if $P(r)$ contains a piece-wise linear region from r_1 to r_2, with slope greater than P_{max}, then it is always more energy efficient to operate at r_1, the lowest possible utilization. This is the direct opposite of our result earlier. We can intuitively understand why operating at r_1 makes sense here. As the slope of $P(r)$ approach infinity, i.e. as $P(r)$ approach a vertical line between r_1 and r_2, a marginal drop in the utilization from r_2 to r_1 can decrease power consumption significantly while keeping the job duration near constant. In other words, we reduce energy consumption for free. To the best of our knowledge, no real life systems exhibit this kind of power-utilization characteristic. But if one day such systems exist, then default operation should be at as low utilization as possible, subject to job duration constraints.

One can object that power consumption is not linear. However, recent measurements in [14] showed that our models are quite sound. The authors used a multi-variable linear model to predict system power with root mean square error of approximately 2W for a 200W system. The model included the power consumption of CPU, memory, disk, and network. The data also showed that for CPUs, if we take r to mean the number of instructions per second, then the system power consumption is exactly as shown in Figure 19 - linear power increase with large idle power offset. Using a less detailed analysis, the authors also arrived at our conclusion, that if work rate is proportional to resources used, we should always run our workloads as fast as possible.

Our own measurements from the previous sections both support the analysis here and reveal its limitations. For a MapReduce cluster, we can interpret r to mean the fraction of workers used. When the number of workers is constant, our model suggests that whatever techniques we use to decrease the job duration would also bring about decreased energy consumption. We find data supporting this tight connection between job duration and energy consumed in Figure 7 for different HDFS replication levels and Figure 10 for different HDFS block size.

In contrast, when the number of workers varies, r varies. Our model suggests that we should always use the maxinum number of workers. However, our discussions for sort (Figure 3), nutch (Figure 4), slow nodes (Table 1), and intermitten job arrival rates (Section 6.2) contradict this result. The reason is that our model assumes that work rate is proportional to r. In the counter-examples we identified, the work rate is not proportional to the number of workers. This behavior is due to sub-optimal scaling in sort, a scaling bottleneck in nutch, a performance penalty for slow nodes, and a work rate capped by the job arrival rate.

If the work rate behavior is quantified, we still have a valid framework of looking at energy efficiency as the ratio of power and work rate, even if we have non-linear $P(r)$ and $R(r)$ functions. The behavior of $E(r)$ would be more complicated, and we have to optimize across the entire operating region from r_l to r_u.

We encounter further complications when we change the scope of the system under analysis. The meaning of r, P_{max}, and P_{idle} depends on whether we are talking about a single machine, a cluster, or an entire datacenter. The insights we get from the model would also be different. For example, when we are talking about a single machine, P_{max} would be the machine's power rating. Our model here highlights the importance of low-power hardware - when P_{max} is decreased, the entire curve for $P(r)$ shifts vertically lower. When we are talking about a cluster, r would be the number of machines. Our model highlights the importance of system scalability and intelligent provisioning according to load - without them, the graph for $R(r)$ would not be proportional. When we are talking about a datacenter, P_{idle} would be the total facility power excluding the IT equipment. Figure 19 highlights the importance of a low PUE - the closer the PUE is to 1, the lower the P_{idle} and the vertical intercept for $P(r)$. The bottomline of our analysis here is that achieving traditional system design goals would benefit both time efficiency and energy efficiency.

7. RECOMMENDATIONS

Our findings are summarized into the following recommendations for MapReduce/Hadoop operators:

- Measure the duration, energy, and power of common MapReduce/Hadoop tasks to find out if they are scalable to more workers.
- Ensure that small sized jobs are run on an appropriately small number of workers.
- Ensure that HDFS block sizes increases with increased data size.

- Consider using the decision process in Section 6.2 to provision the number of workers.

- If Hadoop is deployed concurrently with a reliable storage system separate from HDFS, then we should always operate with HDFS replication of 1, even when we incorporate the cost of data staging in and out of HDFS, per the analysis in Section 6.3.

- Measure the power-utilization and work rate-utilization characteristics for machines in the cluster, and use the framework in Section 6.4 to optimize the level of utilization.

- Continue to focus on all techniques for performance improvements, since energy efficiency is equivalent to time efficiency when work rate is proportional to the resources used.

For MapReduce/Hadoop designers, we make the following suggestions:

- Improve the HDFS replication process. The current "chain" of replication play a significant part in the high variability in jobs involving large amounts of output. A lower HDFS replication level is not possible when data durability is important, per the analysis in Section 6.3.

- Decrease the job coordination overhead. As indicated in Section 5.5, overhead prevents small jobs from scaling to more workers.

- Improve data locality for jobs. As indicated in Section 5.5, increased data locality is as important as increased parallelism by adding more workers.

- Investigate ways to assign less work to slow nodes, or to remove them from the cluster when they impose more penalty than the speedup they contribute.

For researchers working on MapReduce/Hadoop energy efficiency, we propose the following work items:

- Repeat our experiments on clusters running different machines, in particular machines that are designed to be energy efficient, e.g. machines with ATOM processors.

- Repeat our experiments on other MapReduce/Hadoop workloads, such as gridmix [5].

- Investigate the effect of changing other Hadoop parameters, in particular IO parameters like HDFS block size, number of parallel reduce copies, sort memory limit, and file buffer size.

- Compare the usefulness of MapReduce as a benchmark vis-a-vis established benchmarks such as SPECPower.

8. CLOSING REMARKS

We began our work with the goal to add energy as another performance metric to consider in the design space for computer systems and Internet datacenters. We concluded our work with the thought that energy efficiency and traditional time efficiency are fundamentally different performance metrics for the same thing. Whatever that thing is, we do not yet have a well articulated language to talk about it. We believe the natural continuation of our work is the development of MapReduce/Hadoop benchmarks, perhaps across a bundle of performance metrics at first. In the long term, it would be a tremendous contribution to develop a way to talk about computer systems or Internet datacenters productivity that unifies the seemingly disparate concerns currently measured using different metrics. Taking a high level view, the early single-machine SPEC benchmarks are the first step towards this goal, the PUE metric is the second step specific for Internet datacenters. We believe our work is making incremental progress towards understanding computer systems productivity in general. We encourage other researchers to continue in the same direction, whether they come from energy perspectives, or perspectives focused more on traditional performance.

9. ACKNOWLEDGMENTS

Thanks to the rest of the RAD Lab team for their frequent and insightful suggestions. Special thanks to our IT staff who provided tremendous help with cluster operation and power meter management.

10. REFERENCES

[1] Annual energy review 2008. Energy Information Administration, U.S. Department of Energy, June 2009.

[2] Energy star enterprise server specification. United States Environmental Protection Agency, http://www.energystar.gov/index.cfm?c=new_specs.enterprise_servers, April 2009.

[3] Energy star enterprise storage specification. United States Environmental Protection Agency, http://www.energystar.gov/index.cfm?c=new_specs.enterprise_storage, under development, April 2009.

[4] Green Grid data center efficiency metrics: PUE and DCIE. White Paper, The Green Grid, December 2008.

[5] Gridmix. HADOOP-HOME/src/benchmarks/gridmix in all recent Hadoop distributions.

[6] Hadoop. http://hadoop.apache.org.

[7] Hadoop Power-By Page. http://wiki.apache.org/hadoop/PoweredBy.

[8] International energy annual 2006. Energy Information Administration, U.S. Department of Energy, June 2008.

[9] Nutch. http://lucene.apache.org/nutch/.

[10] SPECpower_ssj2008. Standard Performance Evaluation Corporation, http://www.spec.org/power_ssj2008/, April 2009.

[11] J. Ammer and J. Rabacy. The energy-per-useful-bit metric for evaluating and optimizing sensor network physical layers. In *Sensor and Ad Hoc Communications and Networks, 2006. SECON '06. 2006 3rd Annual IEEE Communications Society on*, volume 2, pages 695–700, Sept. 2006.

[12] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[13] C. Belady. In the data center, power and cooling costs more than the IT equipment it supports. *Electronics Cooling Magazine*, 13(1):24–27, February 2007.

[14] S. Dawson-Haggerty, A. Krioukov, and D. Culler. Power optimization - a reality check. Submitted to HotPower 2009, Workshop on Power Aware Computing and Systems.

[15] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[16] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 13–23, New York, NY, USA, 2007. ACM.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[18] Google. Data center efficiency measurements. The Google Blog, 2009.

[19] J. Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3), September 2008.

[20] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 315–326, Washington, DC, USA, 2008. IEEE Computer Society.

[21] M. Manos and C. Belady. What is your PUE strategy. *The Power of Software blog*, July 2008.

[22] R. Miller. Microsoft: PUE of 1.22 for data center containers. Data Center Knowledge, October 2008.

[23] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 365–376, New York, NY, USA, 2007. ACM.

[24] S. Shankland. Google uncloaks once secret server. *cnet news*, April 2009.