Acknowledgement

We would like to thank Michael Armbrust, Rodrigo Fonseca, George Porter, David Patterson and Armando Fox, as well as our industrial contacts at Yahoo!, Facebook and Amazon, for their help with this work. This research was supported by California MICRO, California Discovery, the Natural Sciences and Engineering Research Council of Canada, as well as the following Berkeley Reliable Adaptive Distributed systems lab sponsors: Sun, Google, Microsoft, HP, Cisco, Oracle, IBM, NetApp, Fujitsu, Siemens, and VMWare.

Improving MapReduce Performance in Heterogeneous Environments

Matei Zaharia Andy Konwinski Anthony D. Joseph Randy Katz Ion Stoica
UC Berkeley

Abstract

MapReduce is emerging as an important programming model for large-scale data-parallel applications such as web indexing, data mining, and scientific simulation. Hadoop is an open-source implementation of MapReduce enjoying wide adoption and is often used for short jobs where low response time is critical. Hadoop's performance is closely tied to its task scheduler, which implicitly assumes that cluster nodes are homogeneous and tasks make progress linearly, and uses these assumptions to decide when to speculatively re-execute tasks that appear to be stragglers. In practice, the homogeneity assumptions do not always hold. An especially compelling setting where this occurs is a virtualized data center, such as Amazon's Elastic Compute Cloud (EC2). We show that Hadoop's scheduler can cause severe performance degradation in heterogeneous environments. We design a new scheduling algorithm, Longest Approximate Time to End (LATE), that is both simple and highly robust to heterogeneity. LATE can improve Hadoop response times by a factor of 2 in 200-node clusters on EC2.

1 Introduction

Today's most popular computer applications are Internet services with millions of users. The sheer volume of data that these services process has led to interest in parallel processing on commodity clusters. The leading example is Google, which uses its MapReduce framework to process 20 petabytes of data per day [1]. Other Internet services, such as e-commerce sites, social networks, communication services and content providers, also cope with enormous volumes of data. These services generate clickstream data from millions of users every day, which is a potential gold mine for understanding access patterns and increasing ad revenue. Furthermore, for each user interaction, a web application generates one or two orders of magnitude more data in system logs, which are the main tool that developers and operators have for diagnosing problems with the system.

The MapReduce model popularized by Google provides a very attractive framework for ad-hoc parallel processing on arbitrary data. MapReduce breaks a computation into small tasks that run in parallel on different machines, and scales easily to very large clusters of inexpensive commodity computers. Its popular open-source implementation, Hadoop [4], was developed primarily by Yahoo, where it processes hundreds of terabytes of data on at least 10,000 cores [11], and is now used by other companies, including Facebook, Amazon, and Last.fm, and the New York Times [10]. Research groups at Cornell, Carnegie Mellon, University of Maryland and PARC are also starting to use Hadoop for both web data and non-data-mining applications, like seismic simulation and natural language processing [10, 12].

A key benefit of MapReduce is that it automatically handles failures, hiding the complexity of fault-tolerance from the programmer. If a node crashes, MapReduce reruns its tasks on a different machine. Equally importantly, if a node is available but is performing poorly, a condition that we call a *straggler*, MapReduce runs a *speculative copy* of its task (also called a "backup task") on another machine to finish the computation faster. Without this mechanism of *speculative execution*[1], a job would be as slow as its slowest sub-task. Stragglers can arise for many reasons, including faulty hardware and misconfiguration. Google has noted that speculative execution can improve job response times by 44% [1].

In this work, we address the problem of how to robustly perform speculative execution to maximize performance. Hadoop's scheduler starts speculative tasks based on a simple heuristic comparing each task's progress to the average progress. Although this heuristic works well in homogeneous environments where stragglers are obvious, we show that it can lead to severe

[1]Not to be confused with speculative execution at the OS or hardware level such as in Speculator [20].

performance degradation when its underlying assumptions are broken. An especially compelling environment where Hadoop's scheduler is inadequate is a virtualized data center. Virtualized "utility computing" environments, such as Amazon's Elastic Compute Cloud (EC2) [2], are becoming an important element in an organization's data processing toolbox because large numbers of virtual machines can be rented by the hour at lower costs than managing a data center year-round (EC2's current cost is $0.10 per CPU hour). For example, the New York Times rented 100 virtual machines over a day to convert 11 million scanned articles to PDFs [13]. These environments offer an economic advantage – the ability to own large amounts of compute power only when needed – but they come with the caveat of having to run on virtualized resources with potentially uncontrollable variance. Furthermore, we expect heterogeneous environments to become the common case even in private data centers as organizations often use multiple generations of hardware. Also, private data centers are starting to use virtualization to simplify administration, consolidate servers, and provide more efficient scheduling and resource utilization. We have observed that Hadoop's assumptions of homogeneity lead to incorrect and often excessive speculative execution in heterogeneous environments, and can even degrade performance below that obtained without speculative execution. Sometimes as many as 80% of tasks are speculatively executed, and throughput is 50% less than that of a system with speculative execution disabled.

Naïvely, one might expect speculative execution to be a simple matter of duplicating tasks that are sufficiently slow. In reality, it is a complex issue for several reasons. First, speculative tasks are not free – they compete for certain resources, such as the network, with other running tasks. Second, choosing a node on which to launch a speculative task is as important as choosing which task to launch. Third, in a heterogeneous environment, it may be difficult to differentiate between nodes that are slightly slower than the mean and stragglers.

Starting from first principles, we design a simple algorithm for speculative execution that is robust to heterogeneity and highly effective in practice. We call our algorithm LATE for Longest Approximate Time to End. LATE is based on three principles: prioritizing tasks to speculate, selecting fast nodes to run on, and capping speculative tasks to prevent thrashing. We show that our scheduler can improve response time of MapReduce jobs by a factor of 2 in large clusters on EC2.

The rest of the paper is organized as follows. Section 2 describes Hadoop's current scheduler and the assumptions it makes. Section 3 shows how these assumptions break in heterogeneous environments. Section 4 introduces our new scheduler, LATE, and explains how it addresses these issues. Section 5 validates our claims about

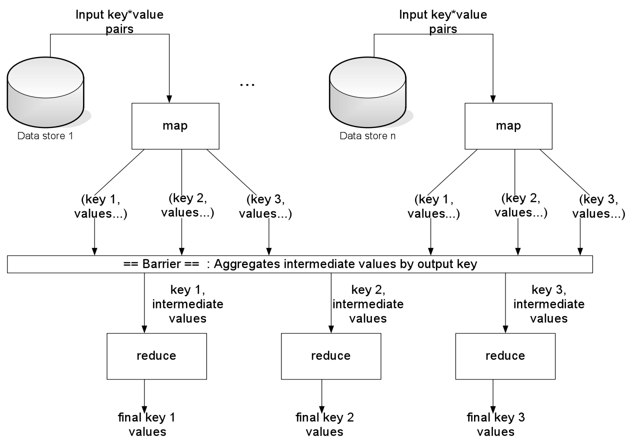

Figure 1: A MapReduce computation. Image from [14].

heterogeneity in virtualized environments through measurements of EC2 and evaluates the gain from LATE in a variety of experiments. Section 7 presents related work. Finally, we conclude in Section 8.

2 Background: Scheduling in Hadoop

In this section we describe the mechanism used in Hadoop to distribute work across a cluster. We identify assumptions made by the scheduler that hinder its performance. These motivate our LATE scheduler, which can outperform Hadoop's by a factor of 2.

Hadoop's implementation of MapReduce closely resembles Google's [1]. There is a single *master* controlling a number of *slaves*. The input file, which resides on a distributed filesystem throughout the cluster, is broken into even sized *chunks*. Hadoop divides each MapReduce job into a series of *tasks*. Each chunk of input is first processed by a *map* task, which outputs a series of key-value pairs generated by a user-defined map function. Map outputs are split into buckets based on key. When all maps have finished, *reduce* tasks apply a reduce function to the map outputs for each key. Figure 1 shows a diagram of a MapReduce computation.

Hadoop runs several maps and reduces concurrently on each slave – two of each by default – to overlap computation and I/O. Each slave tells the master when it has an empty task slot. The scheduler then assigns it a task.

The goal of speculative execution is to minimize a job's *response time*. Response time is most important for short jobs where a user wants an answer quickly, such as queries on log data for debugging, monitoring and business intelligence. Short jobs are a major use of MapReduce. For example, the average MapReduce job at Google in September 2007 was 395 seconds long [1]. Systems designed for SQL-like queries on top of

MapReduce, such as Sawzall [15] and Pig [16], underline the importance of MapReduce for query processing. Response time is also clearly important in a pay-by-the-hour environment such as EC2. Speculative execution is less useful in long jobs, because only the last wave of tasks is affected, and may be inappropriate for batched jobs if throughput is the only metric of interest, because speculative tasks imply wasted work. In our work, we focus on short jobs.

2.1 Hadoop's Scheduling Algorithm

When a node has an empty task slot, Hadoop chooses a task for it from one of three categories. First, if any task has failed, it is given highest priority. This is done to detect when a task fails repeatedly due to a bug and stop the job. Second, unscheduled tasks are considered. For maps, tasks with data local to the node are chosen first. Finally, Hadoop looks for a task to speculate on.

To select speculative tasks, Hadoop monitors task progress using a *progress score*, which is a number from 0 to 1. For a map, the score is the fraction of input data read. For a reduce task, the execution is divided into three phases, each of which accounts for 1/3 of the score:

- The *copy phase*, when the task is copying outputs of all maps. In this phase, the score is the percent of maps that output has been copied from.
- The *sort* phase, when map outputs are sorted by key. Here the score is the percent of data merged.
- The *reduce* phase, when a user-defined function is applied to the map outputs. Here the score is the percent of data passed through the reduce function.

For example, a task halfway through the copy phase has a score of $1/2 \times 1/3 = 1/6$. A task halfway through the reduce phase scores $1/3 + 1/3 + (1/2 \times 1/3) = 5/6$.

Hadoop looks at the average progress of each category of tasks (maps and reduces) to define a *threshold* for speculative execution: When a task's progress is less than the average for its category minus 0.2, and the task has run for at least one minute, it is marked as a straggler. All tasks beyond the threshold are considered "equally slow," and ties between them are broken by data locality. The scheduler also ensures that at most one speculative copy of each task is running at a time.

Although a metric like progress rate would make more sense than absolute progress for identifying stragglers, the threshold in Hadoop works reasonably well in homogenous environments because tasks tend to start and finish in "waves" at roughly the same times and because speculation only starts when the last wave is running.

Finally, when running multiple jobs, Hadoop uses a FIFO system where the earliest submitted job is asked for a task to execute, then the second, etc. There is also a priority system for putting jobs into higher-priority queues.

2.2 Assumptions in Hadoop's Scheduler

Hadoop's scheduler makes several implicit assumptions:

1. Nodes can perform work at roughly the same rate.
2. Tasks progress at a constant rate throughout time.
3. There is no cost to launching a speculative task on a node that would otherwise have an idle slot.
4. A task's progress score is roughly equal to the fraction of its total work that it has done. Specifically, in a reduce task, the copy, reduce and merge phases each take 1/3 of the total time.
5. Tasks tend to finish in waves, so a task with a low progress score is likely a slow task.
6. Different tasks of the same category (map or reduce) require roughly the same amount of work.

As we shall see, assumptions 1 and 2 break down in a virtual cluster due to heterogeneity. Assumptions 3, 4 and 5 can actually break down in a homogeneous cluster as well, and may cause Hadoop's scheduler to perform poorly in in non-virtualized clusters too. In fact, Yahoo! disables speculative execution on some jobs because it degrades performance, and monitors machines through other means. Facebook disables speculation for reduce tasks [26].

Assumption 6 is inherent in the MapReduce paradigm, so we do not address it in this paper. Tasks in MapReduce should be small, otherwise a single large task will slow down the entire job. In a well-behaved MapReduce job, the separation of input into equal chunks and the division of the key space among reducers ensures roughly equal amounts of work. If this is not the case, launching a few more speculative tasks is not harmful as long as obvious stragglers are also detected.

3 How the Assumptions Break Down

3.1 Heterogeneity

The first two assumptions in Section 2.2 are about homogeneity: Hadoop assumes that any sufficiently slow node is a straggler. However, nodes can be heterogeneous for other reasons. In a non-virtualized data center, there may be multiple generations of hardware. As organizations expand their data centers, this is a growing concern. In a virtualized data center where multiple virtual machines run on each physical host, such as Amazon EC2, co-location of virtual machines may cause heterogeneity. Although virtualization isolates CPU and memory performance, VMs compete for disk and network bandwidth. For example, in EC2, instances use the maximum bandwidth when there is no contention and share bandwidth fairly when there is contention [22]. Contention can come both from other users' virtual machines, in which case it may be transient, and from one's *own* virtual machines if several share the same physical host, especially if they do similar work, as in Hadoop. In Section 5.1, we measured performance differences of 2.5x caused by contention. Note that EC2's bandwidth sharing policy is not inherently harmful – it means that I/O bandwidth can be fully utilized when some VMs do not use it – but it causes a problem in Hadoop.

Heterogeneity seriously impacts Hadoop's scheduler. Because the scheduler uses a fixed threshold for selecting tasks to speculate, *too many* speculative tasks may be launched, taking away resources from useful tasks (assumption 3 is also false). Also, because the scheduler ranks candidates by locality, the *wrong* tasks may be chosen for speculation first. For example, if the average progress was 70% and there was a 1.4x slower task at 50% progress and a 10x slower task at 7% progress, then the 1.4x slower task might be speculated before the 10x slower task if its data is closer to an idle node.

We note that EC2 also provides "large" and "extra large" VM sizes that have lower variance in I/O performance than the default "small" VMs, possibly because they own a full disk. However, small VMs can achieve higher I/O performance per dollar because they use all available bandwidth when no other VMs on the host are using it. Larger VMs also still compete for network bandwidth. Therefore, we focus on optimizing Hadoop on "small" VMs to get the best performance per dollar.

3.2 Other Assumptions

Assumptions 3, 4 and 5 in Section 2.2 are broken on both homogeneous and heterogeneous clusters, and can lead to a variety of failure modes for the scheduler.

Assumption 4, that a task's progress score is roughly equal to its percent completion, can cause incorrect speculation of reducers. In a typical MapReduce job, the copy phase of reduce tasks is the slowest, because it involves all-pairs communication over the network. Tasks quickly complete the other two phases once they have all the map outputs. However, the reduce phase counts as 1/3 of the total progress score, while the other phases count as 2/3. Thus, when the first few reducers in a job finish the copy phase, their progress goes from 1/3 to 1, greatly increasing the average progress. As soon as about 30% of reducers finish, the average progress is roughly $0.3 \times 1 + 0.7 \times 1/3 \approx 53\%$, and now *all* reducers still in the copy phase will be 0.2 behind the average, and an arbitrary set will be speculated upon causing task slots to fill up, and true stragglers to never be speculated. The problem is worse in a heterogeneous environment. If a large fraction of reducers to stay in the copy phase for longer than average, then these reducers will *all* be speculated upon, potentially overloading the network. We have observed this in 900-node runs on EC2, where 80% of reduce tasks were speculated.

Assumption 4 also means that when a task reaches 80% progress, it can never be speculated upon, even if it slows down, because the average progress is always less than 100%.

Assumption 5, that tasks start at roughly the same time and so progress *score* is a good proxy for progress *rate*, can also be wrong. In a MapReduce job, there are typically one or two reducers per node so they can start copying data over the network right away, so the assumption holds for reducers. However, there are potentially tens of mappers per host, one for each data chunk. These tend to finish in waves. Even in a homogenous environment, the waves get more spread out as more mappers run due to variance adding up, so in a long enough job, tasks from different generations will be executing concurrently. In this case, Hadoop will speculate on newer, faster tasks ahead of older, slow tasks that have more total progress.

Lastly, assumption 3 – that speculating tasks on idle nodes is free – breaks down when resources are shared. For example, the network is a bottleneck shared resource in large MapReduce jobs. Also, speculative tasks may compete for disk I/O's in disk I/O-bound jobs. Finally, in a series of jobs, needless speculation reduces throughput without improving response time by occupying nodes that could be running the next job.

4 The LATE Scheduler

We have designed a new speculative task scheduler by starting from first principles and adding features needed to correctly schedule tasks in a real environment.

The primary insight behind our scheduler is the following: We always speculatively execute the task that we think will finish *farthest into the future*, because this task provides the greatest opportunity for a speculative task to overtake the original and save a significant amount of time. We explain how we calculate a task's remaining time based on current progress below. We call our strategy LATE, for Longest Approximate Time to End. Intuitively, this greedy policy would be optimal if nodes ran at consistent speeds and if there were no costs associated with launching a speculative task on an otherwise idle node.

Different methods of estimating time left can be plugged into LATE. We currently use a simple heuristic that we found works well in practice: We estimate the *progress rate* for each task as (progress score) / (execution time), and then estimate the time left as (1 - progress) / (progress rate). This assumes that tasks make progress at a roughly uniform rate, but also works well when the earlier phases of the task take longer than the last phases. There are cases where this heuristic can fail, which we describe later, but it is effective in typical Hadoop jobs.

To really get the best chance of beating the original task with the speculative task, we should only launch speculative tasks on *fast nodes*, so as to avoid stragglers. We do this through a simple heuristic – don't launch speculative tasks on nodes that are below some threshold, *SlowNodeThreshold*, of total work performed (sum of progress scores for all succeeded and in-progress tasks on the node). This heuristic leads to better scheduling decisions than assigning a speculative task to any idle node. Another option would be to support more than one speculative copy of each task, but this wastes resources needlessly.

Finally, to handle the fact that speculative tasks cost resources, we augment the algorithm with two heuristics:

- A cap on the number of speculative tasks that can be running at once, which we denote *SpeculativeCap*.
- A *SlowTaskThreshold* that a task's progress rate is compared against to determine whether it is "slow enough" to be launched speculatively.

In summary, the LATE algorithm works as follows:

- If a task slot becomes available and there are less than *SpeculativeCap* speculative tasks running:
  - Ignore the request if the node's total progress is below *SlowNodeThreshold*.
  - Rank currently running, non-speculatively executed tasks by estimated time left.
  - Launch a copy of the highest-ranked task with progress rate below *SlowTaskThreshold*.

Like Hadoop's scheduler, we also wait until a task has run for 1 minute before evaluating it for speculation.

In practice, we have found that a good choice for the three parameters to LATE are to set the *SpeculativeCap* to 10% of available task slots and set the *SlowNodeThreshold* and *SlowTaskThreshold* to the 25th percentile of node progress and task progress rates respectively. We use these settings in our evaluation.

Finally, we note that unlike Hadoop's scheduler, LATE does not take into account data locality for launching speculative map tasks, though this is a potential extension. We assume that because most maps are data-local, network utilization during the map phase is low, so it is fine to launch a speculative task on a fast node that does not have a local copy of the data. Locality statistics available in Hadoop validate this assumption.

4.1 Advantages of LATE

The LATE algorithm has several advantages. First, it is agnostic to node heterogeneity, because it will relaunch only a small number of the slowest tasks. LATE prioritizes among the slow tasks to ensure that only the slowest get speculatively executed. LATE also caps the number of speculative tasks to limit contention for shared resources and avoid thrashing. In contrast, Hadoop's native scheduler has a fixed threshold, beyond which all tasks that are "slow enough" have an equal chance of being launched. This fixed threshold can cause excessively many speculative tasks to be launched.

Second, LATE also takes into account node heterogeneity when deciding where to run a speculative task. In contrast, the native scheduler in Hadoop assumes that any node that finishes a task and has a free slot is likely to be a fast node (i.e., that slow nodes will almost never finish tasks), so it schedules tasks on all idle nodes without discriminating on any property except data locality.

Third, a task's progress score is no longer assumed to be linearly correlated with its percent completion. We do use progress rate in estimating time late, but we always choose the tasks with the lowest progress rate, rather than all tasks with a progress rate below some threshold. This also means that tasks are allowed to slow down occasionally and make progress sporadically. LATE also allows alternative methods of estimating task completion type to be employed if they perform better.

Fourth, because LATE uses estimated time left rather than heuristics based on fixed thresholds, means, and absolute progress scores, it is immune to the failure modes caused by assumptions 4 and 5 in Hadoop's scheduler. In general, LATE is more robust than algorithms based on thresholds, because it uses a cap on speculative tasks as well as percentile-based thresholds for identifying slow nodes and slow tasks.

As a concrete example of how LATE improves over Hadoop's scheduler, consider the reduce example in Section 3.2, where assumption 4 (that progress score represents percent complete) is violated and all reducers still in the copy phase fall below the speculation threshold as soon as a few reducers finish their work. Hadoop's native scheduler would speculate arbitrary reduces, missing true stragglers and potentially starting too many speculative tasks if many reducers stay slow for a period of time. In contrast, LATE would first start speculating the reducers with the slowest copy phase, which are probably the true stragglers, and would stop speculating new reducers once it has reached the *SpeculativeCap*, ensuring that the network does not get overloaded.

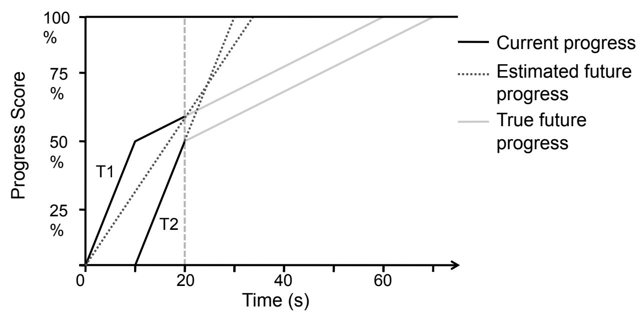

Figure 2: A scenario where LATE estimates finish orders incorrectly.

4.2 Estimating Finish Times

At the beginning of Section 4, we said that we estimate the time left for a task based on the progress score provided by Hadoop, as (1 - progress) / (progress rate). Although this heuristic works well in practice, we wish to point out that there are situations in which it can backfire, and this heuristic might estimate that a task that has made *more* progress than another identical task which will finish *later*. Because this situation does not occur in typical MapReduce jobs (as motivated below), we have used this progress rate heuristic for our evaluation of LATE in this paper. We explain this misestimation here because it is an interesting and subtle factor in scheduling based on rate of progress. In future work, we plan to evaluate more sophisticated methods of estimating finish times.

To see how the progress rate heuristic might backfire, consider a task T that has two phases in which it runs at different rates. Suppose the task makes progress at 5% points per second in the first phase, up to a total progress score of 50%, and then slows down to 1% per second in the second phase. Thus the task spends 10 seconds in the first phase and 50 seconds in the second phase, or 60s in total. Now suppose that we launch two copies of task T, say T1 and T2, one at time 0 and one at time 10, and that we check their progress rates at time 20. Figure 2 illustrates this scenario. At time 20, T1 will have finished its first phase and be one fifth through its second phase, so its total progress score will be 60%, and its progress rate will be $60\%/20\text{s} = 3\%/\text{s}$. Meanwhile, T2 will have just finished its first rate, and its progress rate will be $50\%/10\text{s} = 5\%/\text{s}$. The estimated time left for T1 will be $(100\% - 60\%)/(3\%/\text{s}) = 13.3\text{s}$. The estimated time left for T2 will be $(100\% - 50\%)/(5\%/\text{s}) = 10\text{s}$. Therefore our heuristic will say that T1 will finish after T2, while in reality T2 will finish last.

This situation arises because the task's progress rate slows down throughout its lifetime and is not linearly related to actual progress. In fact, if the task *sped up* in its second phase instead of slowing down, there would be no problem – we would correctly estimate that tasks in their first phase have a longer amount of time left, so the estimated *order* of finish times would be correct, but we would be wrong about the exact amount of time left. The problem in this example is that the task slows down in its second phase, so "younger" tasks seem faster.

Fortunately, this situation does not arise in typical MapReduce jobs in Hadoop. Map tasks' progress is based on the number of records they have processed, so it is always linearly related to running time. Reduce tasks are typically slowest in their first phase – the copy phase, where they must read all map outputs over the network – so they fall into the "speeding up" category above. However, if there is a MapReduce job where some of the later phases of a reduce task are slower than the first, it would be possible to design a more complex heuristic that looks at a task's progress rate in the current phase and compares it with other tasks. We have not done this yet to keep our algorithm simple. We plan to investigate finish time estimation more carefully in future work.

5 Evaluation

We began our evaluation by measuring the effect of contention on performance in EC2, to validate our claims that contention causes heterogeneity. We then ran a suite of experiments evaluating the LATE scheduler in two environments: large clusters on EC2, and a local virtualized testbed.

Although we began our EC2 tests by measuring heterogeneity in the production environment on EC2, we were assigned by Amazon to a separate test cluster when we ran our scheduling tests. Amazon moved us to this test cluster because our tests were exposing a scalability problem in the network virtualization software on their production cluster that was causing connections between our instances to fail intermittently. The test cluster had a patch for this problem. Although fewer clients were contending on the test cluster, we simulated heterogeneity there by occupying almost all the virtual machines in one location – 106 physical hosts, on which we placed 7 or 8 VMs each – and creating contention by using several VMs from each physical host. We chose the distributions of VMs per host to match those observed in the production cluster. In summary, although our results are from a test cluster, they simulate the contention levels seen in production while letting us operate in a more controlled environment. The EC2 results are also consistent with those from our local testbed.

5.1 Measuring Heterogenity on EC2

Virtualization technology can isolate CPU and memory performance effectively between VMs. However, as explained in Section 3.1, heterogeneity can still arise because I/O devices (disk and network) are shared between VMs. On EC2, virtual machines get the full available bandwidth when there is no contention, but are reduced to fair sharing when other VMs on the physical host are doing I/O [22]. We measured the effect of contention on raw disk I/O performance as well as application performance in Hadoop. We saw a difference of 2.5-2.7x between loaded and unloaded machines.

We note that our examples of the effect of load are in some sense extreme, because in practice, EC2 seems to try to place a user's virtual machines on different physical hosts. For example, when we allocated 200 or fewer virtual machines, they were all placed on different physical hosts. Our results are also inapplicable to CPU and memory-bound workloads. However, the results are relevant to users running Hadoop at large scales on EC2, because these users will likely have co-located VMs (as we did) and Hadoop is an I/O-intensive workload.

5.1.1 I/O Performance Heterogeneity

In the first test, we started a `dd` command that wrote 5000 MB of zeroes from `/dev/zero` to a file in parallel on 871 virtual machines in EC2's production cluster. Because EC2 exhibits a "cold start" phenomenon when a block is first written to – possibly due to expanding a VM's disk allocation and clearing existing data on the disk – we first "cleared" 5000 MB of space on each machine by running `dd` and deleting its output. We captured the timing of each `dd` command.

We used a `traceroute` from each virtual machine to an external URL to figure out which physical machine the VM was on – the first hop from a Xen virtual machine is always the `dom0` or supervisor process for that physical host. Our 871 VMs ranged from 202 that were alone on their physical host up to a packing of 7 VMs per physical host. Table 1 shows our results. We see that average write performance ranged from 62 MB/s for the isolated VMs to 25 MB/s for the VMs that shared a physical host with other VMs.

To validate that the performance was tied to contention for disk resources due to multiple VMs writing on the same host, also tried performing `dd`'s in a smaller EC2 allocation where 200 VMs were assigned to 200 separate physical hosts. In this environment, `dd` performance was between 51 and 72 MB/s for all but three VMs. These achieved 44, 36 and 17 MB/s respectively. We do not know the exact cause of these stragglers. The nodes with 44 and 36 MB/s could be explained by contention with other users' VMs given our previous measurements, but the node with 17 MB/s might be a truly faulty straggler. From these results, we conclude that background load is an important factor in virtual machine I/O performance on EC2, and can reduce I/O performance by a factor of 2.5. We have also observed that stragglers can occur "in the wild" on EC2.

Load Level	VMs	Write Perf (MB/s)
1 VMs/host	202	61.8
2 VMs/host	264	56.5
3 VMs/host	201	53.6
4 VMs/host	140	46.4
5 VMs/host	45	34.2
6 VMs/host	12	25.4
7 VMs/host	7	24.8

Table 1: **EC2 Disk Performance versus VM co-location:** Average write performance versus number of VMs per physical host on EC2. The second column shows how many VMs fell into each load level in our 871-node test.

We also measured `dd` throughput on "large" and "extra-large" EC2 VMs. These VMs have two and four virtual disks respectively, which appear to have independent performance. They achieve 50-60 MB/s performance on each disk. However, a large VM costs 4x more than a small one, and an extra-large costs 8x more. Thus the I/O performance per dollar is comparable to, and often less than, that of small VMs.

5.1.2 Impact of Contention at the Application Level

We also evaluated the hypothesis that background load degrades performance in Hadoop itself. For this purpose, we ran two tests with 100 virtual machines: one where each virtual machine was on a separate physical host that was doing no other work, and one where all 100 machines were packed onto 13 physical hosts, with 7 machines per host. To ensure that the physical machines in the first case were doing no other work, we allocated 800 machines in the EC2 test cluster, which we saw fell on 106 physical hosts, and we chose one VM from each of 100 physical hosts. With both sets of machines, we sorted 100 GB of random data using Hadoop's sort benchmark with speculative execution disabled (this setting achieved the best performance). With isolated virtual machines, the job completed in 408s, whereas with machines packed densely onto physical hosts, it took 1094s. Therefore there is a 2.7x difference in Hadoop performance with a cluster of isolated virtual machines versus a cluster of colocated virtual machines. Also note that in this test, some of the contention may have been for network I/O rather than disk I/O, explaining why the difference is larger than for `dd`.

5.2 EC2 Scheduling Experiments

For each measurement in this section we performed several runs on EC2. Due to the environment's variability and contention, there are some runs with high variance in the results. To address this issue, we used a local cluster where we ran more experiments and had full control over the environment.

We evaluated various scheduling strategies in Hadoop in two environments: a large number of virtual machines on Amazon EC2, and a small testbed of virtual machines on 10 physical hosts in one of our department's clusters.

The EC2 machines are all "small"-size instances each of which has 1.7 GB of memory, 1 virtual core with "the equivalent of a 1.0-1.2 GHz 2007 Opteron or Xeon processor," and 160 GB of space on a hard disk which is potentially shared with at most one other VM co-located on the same physical host [22]. Instead of running in the production EC2 environment, they are in a separate test cluster of 106 physical machines that we had nearly full control over, as described at the beginning of section 5. We allocated 800 virtual machines on this cluster – nearly the full capacity, since each physical machine seems to support at most 8 virtual machines – and used a subset of them for each test.

In all tests, we configured the Hadoop Distributed File System to maintain two replicas of each block, and we configured each machine to run up to 2 mappers and 2 reducers simultaneously (the Hadoop default). We chose the data input sizes for our jobs so that each job would last approximately 5 minutes in order simulate shorter, more interactive job-types as are common for MapReduce style applications [1].

In our first set of experiments, we compared three schedulers – Hadoop's native scheduler, our LATE scheduler, and no speculative execution – in two settings: Heterogeneous but non-faulty virtual machines, chosen by assigning a varying number of VMs to each physical host, and an environment with stragglers.

For our workload, we used primarily the Sort benchmark in the Hadoop distribution, but we also evaluated to other workloads. Sorting is the main benchmark used for evaluating Hadoop optimizations at Yahoo! [26]. Google also uses sort as one of the only two performance benchmarks in [1]. In addition, there are a number of features of sorting which make it a desirable benchmark [25].

To validate our findings, we reproduced these experiments in a scaled-down virtual data center on our own cluster in Section 5.3. 5.3.4.

5.2.1 Scheduling in a Heterogeneous Cluster

For our first set of experiments, we purposefully created a heterogeneous cluster by assigning different numbers of virtual machines to physical hosts within a segregated Amazon EC2 test cluster. The cluster had 106 physical machines, each of which hosted 7 or 8 of our virtual machines. We used 1 to 7 virtual machines on each host, for a total of 243 virtual machines. Table 2 shows the number of virtual machines at each load level. We chose this mix to resemble the allocation we saw for 900 nodes in the production EC2 cluster in Section 5.1.

Load Level	**Hosts**	**VMs**
1 VMs/host	40	40
2 VMs/host	20	40
3 VMs/host	15	45
4 VMs/host	10	40
5 VMs/host	8	40
6 VMs/host	4	24
7 VMs/host	2	14
Total	99	243

Table 2: Load level mix in our heterogeneous cluster.

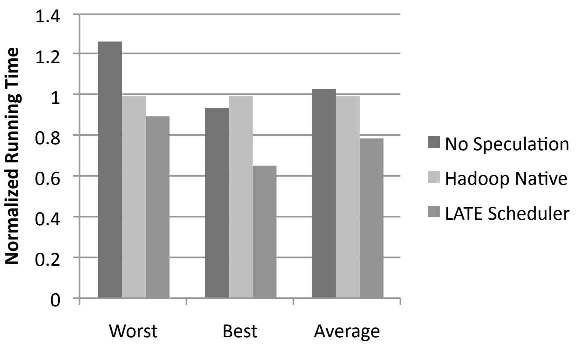

Figure 3: **Running times in heterogeneous cluster:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

As our workload, we used a Sort job on a data set of 128 MB per host, or 30 GB of total data. As in all our experiments, the data was stored with 2 replicas per each block in Hadoop's Distributed File System. Each job had 486 map tasks and 437 reduce tasks (Hadoop leaves some reduce capacity free for speculative and failed tasks). We repeated the experiment 6 times.

Figure 3 compares the response time achieved by each scheduler. Our graphs throughout the Evaluation section show normalized performance against that of Hadoop's native scheduler. We show the worst-case and best-case gain from LATE to give an idea of the range involved. On average, in this heterogeneous cluster experiment, LATE finished jobs about 27% shorter than Hadoop's native scheduler and 31% faster than no speculative execution.

We also evaluated *throughput* by starting three copies of the Sort job simultaneously and measuring the time to

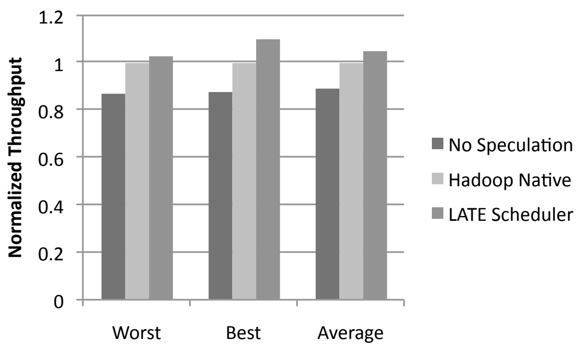

Figure 4: **Throughput in heterogeneous cluster:** Worst, best and average-case throughput (jobs/s) of LATE against Hadoop's scheduler and no speculation.

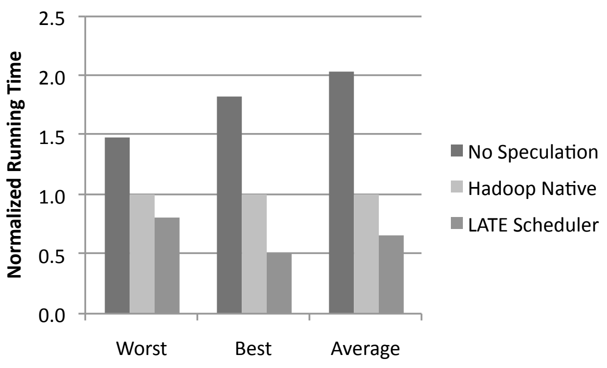

Figure 5: **Running times with stragglers:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

finish all three. The results are shown in Figure 4. We see that LATE performed 5.1% better than Hadoop's scheduler and 18% better than no speculation. As explained in Section 2, as more jobs are submitted to a cluster, no speculation is the best policy for increasing throughput, because it wastes no time at all with speculative task. However, the stragglers in this heterogeneous environment were slow enough that there was still a gain from speculation for three Sort jobs.

5.2.2 Effect of Stragglers

To evaluate the scheduling algorithms in a more interesting and realistic environment, we manually slowed down eight virtual machines in a cluster of 100 VMs to simulate stragglers. The other machines were assigned between 1 and 8 VMs per host, with about 10 in each load level. The stragglers were created by running four CPU-intensive processes (tight loops modifying values in 800 KB arrays) and four disk-intensive processes (`dd` tasks creating large files in a loop) on each of the 8 machines chosen to simulate stragglers. The load was significant enough that disabling speculative tasks caused the cluster to perform 2 to 4 times slower than it did with the LATE scheduler, but not so significant as to render the straggler machines completely unusable. For each run, we sorted 256 MB of data per host, for a total of 25 GB.

Figure 5 shows the results from 4 experiments. On average, LATE finished jobs 58% faster than Hadoop's native scheduler and 220% faster than Hadoop with speculative execution disabled. The gain over native speculative execution could be as high as 93%.

We observe that in some runs, the schedulers' relative performance was similar. There is an element of "luck" involved in these tests that can explain this – if a block's two replicas both happen to be placed on stragglers, then no scheduling algorithm can perform very well because these blocks will be slow to serve. Nonetheless, LATE performs better than the alternatives in all tests.

5.2.3 Differences Across Workloads

To validate our use of the Sort benchmark, we also ran two other workloads, Grep and WordCount, on a heterogeneous cluster with laggards. Grep and WordCount are sample programs in the Hadoop distribution. We used a 204-node cluster with 1 to 8 VMs per physical host. We created eight stragglers using the same combination of CPU and disk-intensive tasks as above.

Grep searches for a regular expression in a text file and creates a file with matches. It then launches a second MapReduce job to sort the matches. We only measured performance of the search job because the sort job was too short for speculative execution to activate (it took less than a minute) and would have resembled our sort benchmarks had it been larger. We applied Grep to 43 GB of text data (repeated copies of Shakespeare's plays), or about 200 MB per host. We searched for the regular expression "the". We repeated the experiment 5 times. The performance results are shown in Figure 6. On average, LATE finished jobs 36% faster than Hadoop's native scheduler and 57% faster than no speculation.

WordCount counts the number of occurrences of each word in a file. We applied WordCount to a smaller data set of 21 GB, or about 100 MB per host. The performance results are shown in Figure 7. We repeated the experiment 5 times. On average, LATE finished jobs 8.5% faster than Hadoop's native scheduler and 179% faster than no speculation.

We observe that the gain from both LATE is smaller in WordCount than in Grep and Sort. This is explained by looking at the workload. Sort and Grep write a significant amount of data over the network and to disk.

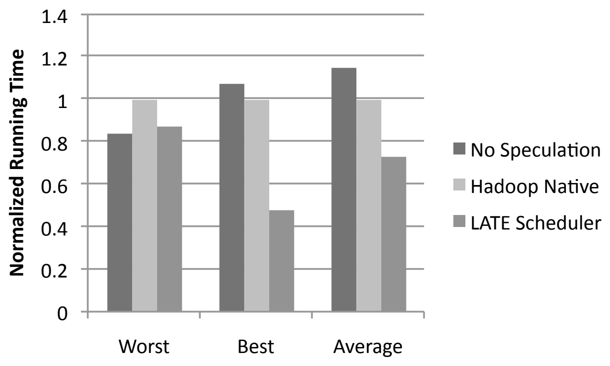

Figure 6: **Grep running times with stragglers:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

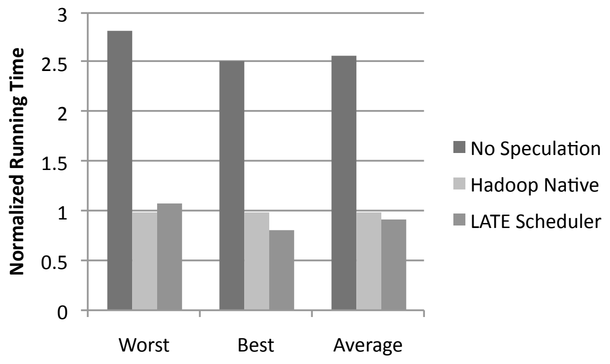

Figure 7: **WordCount running times with stragglers:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

On the other hand, WordCount only sends a small number of bytes to each reducer – a count for each word. Once the maps in WordCount finish, the reducers finish quickly, so its performance is bound by the mappers. The slowest mappers will be those which read data whose only replicas are on straggler nodes, and therefore they will be equally slow with LATE and native speculation. In contrast, in jobs where reducers do a significant amount of work, maps are a smaller fraction of the total time, and LATE has more opportunity to outperform Hadoop's scheduler by speculating reducers more effectively. Nonetheless, in all workloads, we demonstrate that speculation is necessary to mitigate stragglers.

5.3 Local Scheduling Experiments

In order to validate our results from EC2 in a more tightly controlled environment, we also ran a local cluster of 9 physical hosts running Xen virtualization software [21].

Load Level	VMs	Write Perf. (MB/s)	Std.
1 VMs/host	5	52.1	13.7
2 VMs/host	6	20.9	2.7
4 VMs/host	4	10.1	1.1

Table 3: **Local Cluster Disk Performance:** Average write performance and Standard Deviation vs. number of VMs per physical host on our local cluster. The second column shows how many VMs fell into each load level in our local experiments.

Load Level	Hosts	VMs
1 VMs/host	5	5
2 VMs/host	3	6
4 VMs/host	1	4
Total	9	15

Table 4: Load level mix in heterogeneous local cluster.

Our machines were dual-processor, dual-core 2.2 GHz Opteron processors with 4 GB of memory and a single 250GB SATA drive. On each physical machine, we ran one to four virtual machines using Xen, giving each virtual machine 768 MB of memory. While this environment is different from EC2, this appeared to be the most natural way of splitting up the available computing resources among up to four virtual machines per host.

5.3.1 Local Performance Heterogeneity

We first performed a local version of the experiment described in 5.1.1. We started a `dd` command in parallel on each virtual machine which wrote 1GB of zeroes from `/dev/zero` to a file. We captured the timing of each `dd` command and show the averaged results of 10 runs in table 3. We see that average write performance ranged from 52.1 MB/s for the isolated VMs to 10.1 MB/s for the 4 VMs that shared a single physical host.

We witnessed worse disk I/O performance in our local cluster than on EC2 for the co-located virtual machines because our local nodes each have only a single hard disk, thus the machine on which 4 VMs were co-located experienced 4-way disk contention, whereas in the worst case on EC2 8 VMs were contending for 4 hard disks.

5.3.2 Local Heterogeneous Cluster

We next configured the local cluster in a heterogeneous fashion to mimic a VM-to-physical-host mapping one might see in a virtualized environment such as EC2. We chose the number of Virtual Machines per physical host to be appropriate for the size of the hardware we were

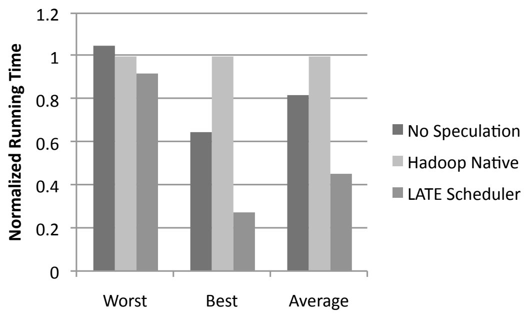

Figure 8: **Running time with heterogeneity:** results of sort benchmark with no speculative execution, Hadoop's native scheduler, and LATE in local heterogeneous cluster.

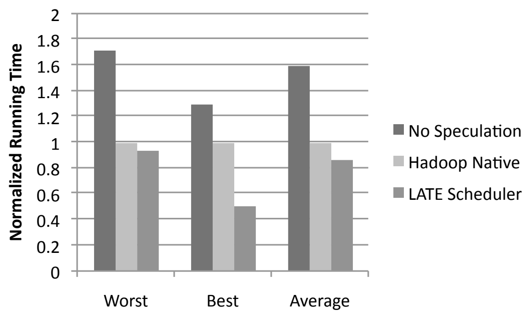

Figure 9: **Running time with stragglers:** results of sort benchmark with: (1) No speculative execution, (2) Hadoop's native scheduler, and (3) LATE in a local heterogeneous cluster with a synthetic straggler.

using, assigning three classes of machines. As Table 4 shows, 5 physical hosts were running a single VM each (one of these machines served as the Hadoop master), two physical hosts were running two VMs each, and one physical host was running 4 VMs. Each virtual machine was configured to operate with 768MB of memory. We then ran our MapReduce sort benchmark, generating 64MB per job, and running each job 5 times. Figure 8 shows the averaged results of 5 runs of the sort benchmark. On average, LATE finished jobs 162% faster than Hadoop's native scheduler and 104% faster than Hadoop with speculative execution disabled. The gain over native speculative execution could be as high as 261%.

5.3.3 Local Stragglers

In order to simulate stragglers in our local control environment, we ran a disk-intensive process (as in section 5.2.2, `dd` tasks creating large files in a loop) on two co-located virtual machines to simulate a poorly performing node. Figure 9 shows the average running times and additionally the worst case and best case performance for LATE. On average, under these simulated straggler conditions, LATE finished jobs 53% faster than Hadoop's native scheduler and 121% faster than Hadoop with speculative execution disabled. The gain over native speculative execution could be as high as 100%.

5.3.4 Differences Across Workloads

Like in our EC2 experiments, we also ran WordCount on our heterogeneous local cluster with synthetically induced stragglers. The results for the normalized performance of no speculative execution, Hadoop native speculative execution, and LATE for WordCount are shown in Figure 10. We see that LATE consistently performs better than the competition, although as in the tests on

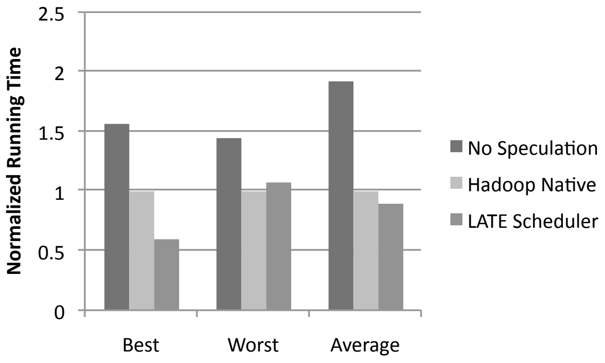

Figure 10: **WordCount running times** with no speculative execution, Hadoop's native scheduler, and LATE scheduling with stragglers.

EC2, the gain in WordCount is less due to the nature of the workload.

6 Discussion

Our work is motivated in part by the decreasing cost of commodity computers and by the availability of open source parallelization frameworks, such as Hadoop, which simplify the task of running distributed applications on thousands of nodes, making it an option for even small organizations.

Having a private cluster offers an organization complete control, but it is also incurs a significant capital and operational cost and provides finite computing resources. In a private cluster, the goal is to engineer solutions that maximize the utilization of the available finite cluster resources and minimize application response times (wall clock running times). Since a private cluster usually evolves over time, it will often contain two or more generations of hardware. The organization may choose to treat the different generations as separate resources, or they may combine them together. The organization may also choose to use virtualization. However, to execute larger and larger tasks, the various physical (and virtual) resources will likely be combined, introducing exactly the types of heterogenity challenges that our results show are addressed by LATE.

If instead, an organization shifts to using a commercial virtualized data center, then they will give up some measure of control and will incur a higher degree of resource heterogeneity and the associated variance. However, computing resources (i.e., compute hours) are no longer a sunk cost (they are now available at an hourly rate) and scale is no longer limited by a local hard cap (the limit is the rate at which organizations like EC2 can grow). Unlike a private cluster, the ultimate limiting factor on scale becomes engineering limitations, such as limited bisection bandwidth in shared nothing parallel computing environments [24]. Instead of simply maximizing finite capacity while minimizing completion time, the organization can now focus on finding the right scale to run a job at in order to satisfy both price and time constraints. An added challenge is that while the average performance in such environments is excellent, there will likely be stragglers that introduce significant performance anomalies. As our results show, by using LATE we are able to effectively leverage speculative execution to mitigate the effects of laggards and resource heterogenity on response time.

We believe that to capitalize on the many benefits that virtualization have to offer–such as simplifying cluster partitioning, sharing, and management–the implications of heterogeneity on distributed applications must be thoroughly understood. Through our experiences with Hadoop in both EC2 and Xen on our local cluster, we have gained a substantial insight into the challenges in this space.

While working on scheduling and speculative execution in heterogeneous environments, we observed three recurring themes that led to improved performance:

- **Tasks are not equal.** If there are multiple slow tasks, then whenever possible a best guess should be made to estimate which task will finish farthest in the future, and that task should be speculatively re-assigned to another node.

- **Compute nodes are not equal.** Whenever possible, speculative tasks should be assigned to fast nodes.

- **Resources are precious.** Thresholds should be used to prevent overloading the system by executing too many speculative tasks.

6.1 Future Work

We have identified several other areas for improving Hadoop's performance in heterogeneous environments. For example, block placement in HDFS currently tries to balance blocks accross nodes, and could be made more intelligent by taking into account each node's disk and network I/O performance. Also, we have begun work on measuring the benefits of prioritizing the allocation of disk and network resources between map and reduce tasks on each MapReduce/HDFS slave, based on preliminary observations that reducers can sometimes compete with mappers and create an effect similar to priority inversion where a few maps finish late and slow down the entire job because most reducers are waiting on them.

In an initial exploration, we performed dynamic analysis of the performance of slaves in the cluster using several machine learning classification algorithms. Given knowledge about the individual capabilities of each slave in the cluster, the scheduler could be more intelligent when deciding how many tasks to simultaneously assign to each slave, and also which nodes to use for assigning speculative tasks. We believe that there would be benefits even with a simple scheduler algorithm that attempts to adapt to the differing capabilities of each slave by using an adaptive task assignment algorithm similar to TCP's slow-start, congestion control and backoff policies.

In Section 4, we define the *SlowNodeThreshold*, *SlowTaskThreshold* and *SpeculativeCap*. One weakness of any hard threshold is that for a mostly homogenous workload, we are likely to misidentify nodes whose performance is only slightly worse than the mean as stragglers and speculative re-execute them. We plan to improve upon this simple heuristic by using a variance threshold instead of a percentile threshold (i.e., a task will be re-executed if its performance is statistically identified as an outlier). We also wish to measure sensitivity of LATE's performance to these thresholds.

In Section 4, we briefly mention that for speculative execution, Hadoop's current scheduler accounts for a node's proximity to data when choosing which node to assign a map task to. However, LATE does not currently account for data proximity. Because the maximum size of such a copy will be one block of the input data (the default is 64MB in both Hadoop and Google's MapReduce), we do not believe that this will have a significant impact on performance. However, all other factors being equal, accounting for such locality in a future revision of LATE may yield a small reduction in running time.

7 Related Work

Much work has been done on the problem of scheduling policies for task assignment to hosts in distributed systems [17] [18]. While our work shares much in common with this work, previous work is focused on independent tasks, while MapReduce tasks are dependent on co-completion. As a result of assuming task independence this class of work does not have to deal with stragglers (i.e. some tasks being slowed down by others who are taking an unusually longer to complete slowing) and thus also does not have to address scheduling speculative re-execution, which is a core element of our contribution. Also, while this work focuses on distributed computing to support backend services in a server client oriented setting (such as a distributed Web server), the required level of distribution and scale is much smaller than that of many Hadoop installations. Because of our use of EC2 and our focus on Hadoop, the scale at which we are testing the effects of heterogeneity–at sizes up to 900 virtual machines on over 400 physical hosts–is larger than any other distributed scheduling work we are aware of. Finally, while this related work discusses in detail the problem of highly variable (i.e. "heavy tailed") workload sizes, we focus instead on highly variable node performance within the cluster.

Our work has some things in common with multiprocessor task scheduling, both with processor heterogeneity [19] and with task duplication when using dependency graphs [29]. Our work differs significantly from such work because we focus on independent tasks, non-shared memory, and a heterogeneous network, while multiprocessor task scheduling work focuses on processing with high intertask communication and a homogeneous network.

Our work also shares some ideas in common with work on "speculative execution" in distributed file systems [20], information gathering [27], and configuration management [28]. However, while this work is focused on guessing along decision branches, speculative execution in Hadoop is focused on guessing the expected execution time for task assignments in a distributed system.

The Google File System and MapReduce were described architecturally and evaluated for end-to-end performance in [3] and [1]. However, [1] only briefly discusses speculative execution and does not explore the algorithms involved in speculative execution nor the implications of highly variable node performance. Our work provides a detailed look at the speculative execution scheduling mechanism.

DataSynapse, Inc. currently holds a patent [23] which details speculative execution for scheduling in a distributed computing platform. The patent references simple statistics similar to those we have discussed for use in the scheduling of speculative tasks. The statistics they use include mean, normalized mean, standard deviation, and a fraction or percentage of waiting vs. pending tasks associated with each active job.

8 Conclusion

We have analyzed the current speculative execution scheduler in the Hadoop MapReduce framework and identified five primary assumptions upon which it is built that assume homogeneity. We have observed and measured he implications of virtualization in distributed systems both at very large scales in Amazon EC2 and in a more controlled local virtual environment. Through such measurements, we have s hown how virtualized environments introduce heterogeneity that violates the assumptions of Hadoop's current speculative execution scheduler. Based on our findings, we proposed a simple, robust scheduling algorithm, LATE, which performs well in a variety of settings.

9 Acknowledgments

We would like to thank Michael Armbrust, Rodrigo Fonseca, George Porter, David Patterson and Armando Fox, as well as our industrial contacts at Yahoo!, Facebook and Amazon, for their help with this work. This research was supported by California MICRO, California Discovery, the Natural Sciences and Engineering Research Council of Canada, as well as the following Berkeley Reliable Adaptive Distributed systems lab sponsors: Sun, Google, Microsoft, HP, Cisco, Oracle, IBM, NetApp, Fujitsu, Siemens, and VMWare.

References

[1] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Communications of the ACM, Volume 51, Issue 1, pp. 107-113, 2008.

[2] Amazon Elastic Compute Cloud: `http://www.amazon.com/gp/browse.html?node=201590011`

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In Proceedings of 19th Symposium on Operating Systems Principles, pp. 29-43, 2003.

[4] Hadoop: `http://lucene.apache.org/hadoop`

[5] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In Proceedings of 4th USENIX Symposium on Networked Systems Design & Implementation, pp. 271-284, 2007.

[6] Nutch: `http://lucene.apache.org/nutch`

[7] Matthew L. Massie, Brent N. Chun and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Parallel Computing, Vol. 30, Issue 7, July, 2004.

[8] F.D. Sacerdoti, M.J. Katz, M.L. Massie, and D.E. Culler. Wide area cluster monitoring with Ganglia. In Proceedings of the 2003 IEEE International Conference on Cluster Computing, pp. 289-298, December 2003.

[9] Hadoop On Demand: `http://hadoop.apache.org/core/docs/r0.16.3/hod.html`

[10] Applications powered by Hadoop: `http://wiki.apache.org/hadoop/PoweredBy`

[11] Yahoo! Launches World's Largest Hadoop Production Application, Yahoo! Developer Network, `http://developer.yahoo.com/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html`

[12] Presentations by Steve Schlosser and Jimmy Lin at the 2008 Hadoop Summit. `http://developer.yahoo.com/hadoop/summit/`.

[13] D. Gottfrid, Self-service, Prorated Super Computing Fun, The New York Times Blog, `http://open.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/`

[14] Figure from slide deck on MapReduce from Google cluster computing and MapReduce course, `http://code.google.com/edu/submissions/mapreduce-minilecture/lec2-mapred.ppt`. Available under Creative Commons Attribution 2.5 License.

[15] R. Pike, S. Dorward, R. Griesemer, S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall Scientific Programming Journal, Vol. 13, No. 4, pp. 227-298, Oct. 2005.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. To appear in ACM SIGMOD 2008, June 2008.

[17] Mor Harchol-Balter, Task Assignment with Unknown Duration. Journal of the ACM , Vol. 49, No. 2, March 2002, pp. 260-288.

[18] M.Crovella, M.Harchol-Balter, and C.D. Murta. Task assignment in a distributed system: Improving performance by unbalancing load. In Measurement and Modeling of Computer Systems, pages 268269, 1998.

[19] B.Ucar, C.Aykanat, K.Kaya, and M.Ikinci. Task assignment in heterogeneous computing systems. Journal of Parallel and Distributed Computing, 66(1):3246, January 2006.

[20] E.B. Nightingale, P.M. Chen, and J.Flinn. Speculative execution in a distributed file system. ACM Trans. Comput. Syst., 24(4):361392, November 2006.

[21] B.Dragovic, K.Fraser, S.Hand, T.Harris, A.Ho, I.Pratt, A.Warfield, P.Barham, and R.Neugebauer. Xen and the art of virtualization. In Proceedings of the ACM Symposium on Operating Systems Principles, October 2003.

[22] Amazon EC2 Instance Types, `http://www.amazon.com/Instances-EC2-AWS/b?ie=UTF8&node=370375011`

[23] James Bernardin, Peter Lee, James Lewis, DataSynapse, Inc. Using Execution statistics to select tasks for redundant assignment in a distributed computing platform Patent number: 7093004, filed Nov 27, 2002, issued Aug 15, 2006

[24] D.Dewitt and J.Gray. Parallel database systems: The future of database processing or a passing fad. ACM SIGMOD Record, Special Issue on Directions for Future Database Research and Development, 19(4), 1990.

[25] G. E. Blelloch, L. Dagum, S. J. Smith, K. Thearling, M. Zagha An evaluation of sorting as a supercomputer benchmark. Nasa Technical Reports, Jan. 29, 1993

[26] Personal communication with the Yahoo! Hadoop team and Joydeep Sen Sarma (Facebook).

[27] G. Barish. Speculative plan execution for information agents. PhD dissertation, University of Southernt California. Dec. 2003

[28] Y. Su, M. Attariyan, J. Flinn AutoBash: improving configuration management with operating system causality analysis. Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pp. 237-250,2007.

[29] S.Manoharan. Effect of task duplication on the assignment of dependency graphs. Parallel Comput., 27(3):257268, 2001.