Improving MapReduce Performance in Heterogeneous Environments

Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, Ion Stoica
University of California, Berkeley
{matei,andyk,adj,randy,stoica}@cs.berkeley.edu

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 highly robust to heterogeneity. LATE can improve Hadoop response times by a factor of 2 in clusters of 200 virtual machines 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 work with 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 websites and social networks, 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 action, a web application generates one or two orders of magnitude more data in system logs, which are the main resource that developers and operators have for diagnosing problems in production.

The MapReduce model popularized by Google is very attractive for ad-hoc parallel processing of arbitrary data. MapReduce breaks a computation into small tasks that run in parallel on multiple machines, and scales easily to very large clusters of inexpensive commodity computers. Its popular open-source implementation, Hadoop [2], was developed primarily by Yahoo, where it runs jobs that produce hundreds of terabytes of data on at least 10,000 cores [4]. Hadoop is also used at Facebook, Amazon, and Last.fm [5]. In addition, researchers at Cornell, Carnegie Mellon, University of Maryland and PARC are starting to use Hadoop for seismic simulation, natural language processing, and mining web data [5, 6].

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 the misbehaving 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 performance degradation when its underlying assumptions are broken. We design an improved scheduling algorithm that reduces Hadoop's response time by a factor of 2.

An especially compelling environment where

[1]Not to be confused with speculative execution at the OS or hardware level for branch prediction, as in Speculator [11].

Hadoop's scheduler is inadequate is a virtualized data center. Virtualized "utility computing" environments, such as Amazon's Elastic Compute Cloud (EC2) [3], are becoming an important tool for organizations that must process large amounts of data, because large numbers of virtual machines can be rented by the hour at lower costs than operating 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 for a day to convert 11 million scanned articles to PDFs [7]. Utility computing environments provide an economic advantage (paying by the hour), but they come with the caveat of having to run on virtualized resources with uncontrollable variations in performance. We also expect heterogeneous environments to become common in private data centers, as organizations often own multiple generations of hardware, and data centers are starting to use virtualization to simplify management and consolidate servers. We observed that Hadoop's homogeneity assumptions lead to incorrect and often excessive speculative execution in heterogeneous environments, and can even degrade performance below that obtained with speculation disabled. In some experiments, as many as 80% of tasks were speculatively executed.

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 the node to run a speculative task on is as important as choosing the task. Third, in a heterogeneous environment, it may be difficult to distinguish between nodes that are slightly slower than the mean and stragglers. Finally, stragglers should be identified as early as possible to reduce response times.

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 LATE can improve the response time of MapReduce jobs by a factor of 2 in large clusters on EC2.

This paper is organized as follows. Section 2 describes Hadoop's scheduler and the assumptions it makes. Section 3 shows how these assumptions break in heterogeneous environments. Section 4 introduces our new scheduler, LATE. Section 5 validates our claims about heterogeneity in virtualized environments through measurements of EC2 and evaluates LATE in several settings. Section 6 is a discussion. Section 7 presents related work. Finally, we conclude in Section 8.

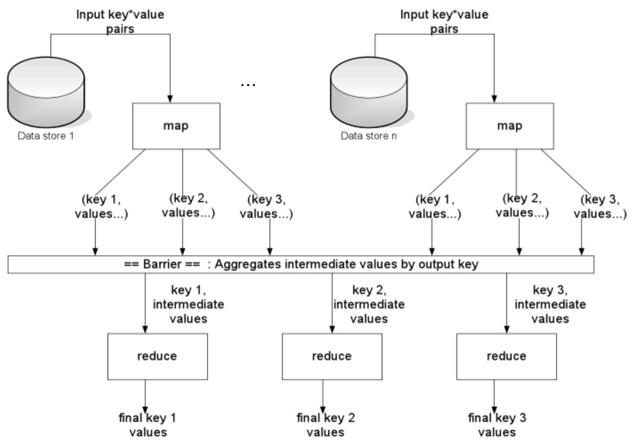

Figure 1: A MapReduce computation. Image from [8].

2 Background: Scheduling in Hadoop

In this section, we describe the mechanism used by Hadoop to distribute work across a cluster. We identify assumptions made by the scheduler that hurt 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* managing a number of *slaves*. The input file, which resides on a distributed filesystem throughout the cluster, is split into even-sized *chunks* replicated for fault-tolerance. Hadoop divides each MapReduce job into a set of *tasks*. Each chunk of input is first processed by a *map* task, which outputs a list 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 list of map outputs with each key. Figure 1 illustrates 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 empty task slots. The scheduler then assigns it tasks.

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 case for MapReduce. For example, the average MapReduce job at Google in September 2007 took 395 seconds [1]. Systems designed for SQL-like queries on top of MapReduce, such as Sawzall [9] and Pig [10], underline the importance of MapReduce for ad-hoc queries. Response time is also clearly important in a pay-by-the-hour environment like EC2. Speculative execution is less useful in long jobs, because only the last wave of tasks is affected, and it may be inappropriate for batch jobs if throughput is the only metric of interest, because speculative tasks imply wasted work. However, even in pure throughput systems, speculation may be beneficial to prevent the prolonged life of many concurrent jobs all suffering from straggler tasks. Such nearly complete jobs occupy resources on the master and disk space for map outputs on the slaves until they terminate. Nonetheless, in our work, we focus on improving response time for short jobs.

2.1 Speculative Execution in Hadoop

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

To select speculative tasks, Hadoop monitors task progress using a *progress score* between 0 and 1. For a map, the progress 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 fetches map outputs.
- The *sort* phase, when map outputs are sorted by key.
- The *reduce* phase, when a user-defined function is applied to the list of map outputs with each key.

In each phase, the score is the fraction of data processed. For example, a task halfway through the copy phase has a progress score of $\frac{1}{2} \cdot \frac{1}{3} = \frac{1}{6}$, while a task halfway through the reduce phase scores $\frac{1}{3} + \frac{1}{3} + (\frac{1}{2} \cdot \frac{1}{3}) = \frac{5}{6}$.

Hadoop looks at the average progress score of each category of tasks (maps and reduces) to define a *threshold* for speculative execution: When a task's progress score 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 speculation only starts when the last wave is running.

Finally, when running multiple jobs, Hadoop uses a FIFO discipline where the earliest submitted job is asked for a task to run, 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 representative of fraction of its total work that it has done. Specifically, in a reduce task, the copy, sort and reduce phases each take about 1/3 of the total time.
5. Tasks tend to finish in waves, so a task with a low progress score is likely a straggler.
6. Tasks in 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 virtualized data center due to heterogeneity. Assumptions 3, 4 and 5 can break down in a homogeneous data center as well, and may cause Hadoop to perform poorly there too. In fact, Yahoo disables speculative execution on some jobs because it degrades performance, and monitors faulty machines through other means. Facebook disables speculation for reduce tasks [14].

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, then launching a few extra 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 detectably slow node is faulty. However, nodes can be slow for other reasons. In a non-virtualized data center, there may be multiple generations of hardware. In a virtualized data center where multiple virtual machines run on each physical host, such as Amazon EC2, co-location of VMs may cause heterogeneity. Although virtualization isolates CPU and memory performance, VMs compete for disk and network bandwidth. In EC2, co-located VMs use a host's full bandwidth when there is no contention and share bandwidth fairly when there is contention [12]. Contention can come from other users' VMs, in which case it may be transient, or from a user's *own* VMs if they do similar work, as in Hadoop. In Section 5.1, we measure performance differences of 2.5x caused by contention. Note that EC2's bandwidth sharing policy is not inherently harmful – it means that a physical host's I/O bandwidth can be fully utilized even when some VMs do not need it – but it causes problems 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 untrue). 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 2x slower task at 35% progress and a 10x slower task at 7% progress, then the 2x slower task might be speculated before the 10x slower task if its input data was available on 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 fully own a disk. However, small VMs can achieve higher I/O performance per dollar because they use all available disk 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.

Assumption 3, that speculating tasks on idle nodes costs nothing, 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 in I/O-bound jobs. Finally, when multiple jobs are submitted, needless speculation reduces throughput without improving response time by occupying nodes that could be running the next job.

Assumption 4, that a task's progress score is approximately 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 map outputs. However, the copy phase counts for only 1/3 of the progress score. Thus, soon after 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 $0.3 \cdot 1 + 0.7 \cdot 1/3 \approx 53\%$, and now *all* reducers still in the copy phase will be 20% behind the average, and an arbitrary set will be speculatively executed. Task slots will fill up, and true stragglers may never be speculated executed, while the network will be overloaded with unnecessary copying. We observed this behavior in 900-node runs on EC2, where 80% of reducers were speculated.

Assumption 5, that progress *score* is a good proxy for progress *rate* because tasks begin at roughly the same time, can also be wrong. The number of reducers in a Hadoop job is typically chosen small enough so that they they can all start running right away, to copy data while maps run. However, there are potentially tens of mappers per node, one for each data chunk. The mappers tend to run in waves. Even in a homogeneous environment, these waves get more spread out over time due to variance adding up, so in a long enough job, tasks from different generations will be running concurrently. In this case, Hadoop will speculatively execute new, fast tasks instead of old, slow tasks that have more total progress.

Finally, the 20% progress difference threshold used by Hadoop's scheduler means that tasks with more than 80% progress can *never* be speculatively executed, because average progress can never exceed 100%.

4 The LATE Scheduler

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

The primary insight behind our algorithm is as follows: 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 copy to overtake the original and reduce the job's response time. We explain how we estimate a task's finish time based on progress score 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 was no cost to launching a speculative task on an otherwise idle node.

Different methods for estimating time left can be plugged into LATE. We currently use a simple heuristic that we found to work well in practice: We estimate the *progress rate* of each task as $ProgressScore/T$, where T is the amount of time the task has been running for, and then estimate the time to completion as $(1 - ProgressScore)/ProgressRate$. This assumes that tasks make progress at a roughly constant rate. 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 also only launch speculative tasks on *fast nodes* – not 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 performance than assigning a speculative task to the first available node. Another option would be to allow 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 with to determine whether it is "slow enough" to be speculated upon. This prevents needless speculation when only fast tasks are running.

In summary, the LATE algorithm works as follows:

- If a node asks for a new task and there are fewer than *SpeculativeCap* speculative tasks running:
  - Ignore the request if the node's total progress is below *SlowNodeThreshold*.
  - Rank currently running tasks that are not currently being speculated 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 values in our evaluation. We have performed a sensitivity analysis in Section 5.4 to show that a wide range of thresholds perform well.

Finally, we note that unlike Hadoop's scheduler, LATE does not take into account data locality for launching speculative map tasks, although 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 robust to node heterogeneity, because it will relaunch only the slowest tasks, and only a small number of tasks. LATE prioritizes among the slow tasks based on how much they hurt job response time. LATE also caps the number of speculative tasks to limit contention for shared resources. 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 tasks to be speculated upon.

Second, LATE takes into account node heterogeneity when deciding *where* to run speculative tasks. In contrast, Hadoop's native scheduler assumes that any node that finishes a task and asks for a new one is likely to be a fast node, i.e. that slow nodes will never finish their original tasks and so will never be candidates for running speculative tasks. This is clearly untrue when some nodes are only slightly (2-3x) slower than the mean.

Finally, by focusing on estimated time left rather than progress rate, LATE speculatively executes only tasks that will improve job response time, rather than any slow tasks. For example, if task A is 5x slower than the mean but has 90% progress, and task B is 2x slower than the mean but is only at 10% progress, then task B will be chosen for speculation first, even though it is has a higher *progress rate*, because it hurts the *response time* more. LATE allows the slow nodes in the cluster to be utilized as long as this does not hurt response time. In contrast, a progress rate based scheduler would always re-execute tasks from slow nodes, wasting time spent by the backup task if the original finishes faster. The use of estimated time left also allows LATE to avoid assumption 4 in Section 2.2 (that progress score is linearly correlated with percent completion): it does not matter how the progress score is calculated, as long as it can be used to estimate the finishing order of tasks.

As a concrete example of how LATE improves over Hadoop's scheduler, consider the reduce example in Section 3.2, where assumption 4 (progress score $\approx$ fraction of work complete) is violated and all reducers in the copy phase fall below the speculation threshold as soon as a few reducers finish. Hadoop's native scheduler would speculate arbitrary reduces, missing true stragglers and potentially starting too many speculative tasks. In contrast, LATE would first start speculating the reducers with the slowest copy phase, which are probably the true stragglers, and would stop launching speculative tasks once it has reached the *SpeculativeCap*, avoiding overloading the network.

4.2 Estimating Finish Times

At the start of Section 4, we said that we estimate the time left for a task based on the progress score provided by Hadoop, as $(1 - ProgressScore)/ProgressRate$. Although this heuristic works well in practice, we wish to point out that there are situations in which it can backfire, and the heuristic might incorrectly estimate that a task which was launched *later* than an identical task will finish *earlier*. Because these situations do not occur in typical MapReduce jobs (as explained below), we have

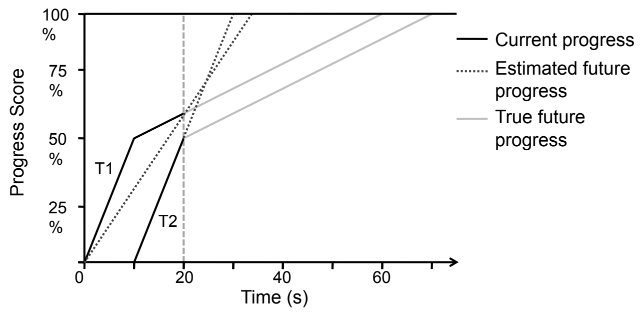

Figure 2: A scenario where LATE estimates task finish orders incorrectly.

used the simple heuristic presented above in our experiments in this paper. We explain this misestimation here because it is an interesting, subtle problem in scheduling using progress rates. 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 that has two phases in which it runs at different rates. Suppose the task's progress score grows by 5% per second in the first phase, up to a total score of 50%, and then slows down to 1% per second in the second phase. 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 the task, 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 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 phase, so 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 take longer to run than T2, while in reality T2 finishes second.

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 frequently arise in typical MapReduce jobs in Hadoop. A map task's progress is based on the number of records it has processed, so its progress is always representative of percent complete. 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 over time" category above.

For the less typical MapReduce jobs 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. Such a heuristic would account for each phase independently when estimating completion time. It would use the the per-phase progress rate thus far observed for any completed or in-progress phases for that task, and for phases that the task has not entered yet, it would use the average progress rate of those phases from other reduce tasks. This more complex heuristic assumes that a task which performs slowly in some phases relative to other tasks will not perform relatively fast in other phases. One issue for this phase-aware heuristic is that it depends on historical averages of per phase task progress rates. However, since speculative tasks are not launched until at least the end of at least one wave of tasks, a sufficient number of tasks will have completed in time for the first speculative task to use the average per phase progress rates. We have not implemented this improved heuristic to keep our algorithm simple. We plan to investigate finish time estimation in more detail 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 evaluated LATE performance in two environments: large clusters on EC2, and a local virtualized testbed. Lastly, we performed a sensitivity analysis of the parameters in LATE.

Throughout our evaluation, we used a number of different environments. We began our evaluation by measuring heterogeneity in the production environment on EC2. However, we were assigned by Amazon to a separate test cluster when we ran our scheduling experiments. Amazon moved us to this test cluster because our experiments were exposing a scalability bug in the network virtualization software running in production that was causing connections between our VMs to fail intermittently. The test cluster had a patch for this problem. Although fewer customers were present on the test cluster, we created contention there by occupying almost all the virtual machines in one location – 106 physical hosts, on which we placed 7 or 8 VMs each – and using multiple VMs from each physical host. We chose our distribution of VMs per host to match that observed in the production cluster. In summary, although our results are from a test cluster, they simulate the level of heterogeneity seen in production while letting us operate in a more controlled environment. The EC2 results are also consistent with those from our local testbed. Finally, when we performed

Environment	Scale (VMs)	Experiments
EC2 production	871	Measuring heterogeneity
EC2 test cluster	100-243	Scheduler performance
Local testbed	15	Measuring heterogeneity, scheduler performance
EC2 production	40	Sensitivity analysis

Table 1: Environments used in evaluation.

the sensitivity analysis, the problem in the production cluster had been fixed, so we were placed back in the production cluster. We used a controlled sleep workload to achieve reproducible sensitivity experiments, as described in Section 5.4. Table 1 summarizes the environments we used throughout our evaluation.

Our EC2 experiments ran on "small"-size EC2 VMs with 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 disk space on potentially shared hard drive [12]. EC2 uses Xen [13] virtualization software.

In all tests, we configured the Hadoop Distributed File System to maintain two replicas of each chunk, 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 run approximately 5 minutes, simulating the shorter, more interactive job-types common in MapReduce [1].

For our workload, we used primarily the Sort benchmark in the Hadoop distribution, but we also evaluated two other MapReduce jobs. Sorting is the main benchmark used for evaluating Hadoop at Yahoo [14], and was also used in Google's paper [1]. In addition, a number of features of sorting make it a desirable benchmark [16].

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, VMs get the full available bandwidth when there is no contention, but are reduced to fair sharing when there is contention [12]. 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 for small allocations, EC2 seems to try to place a user's virtual machines on different physical hosts. 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.

Load Level	VMs	Write Perf (MB/s)	Std Dev
1 VMs/host	202	61.8	4.9
2 VMs/host	264	56.5	10.0
3 VMs/host	201	53.6	11.2
4 VMs/host	140	46.4	11.9
5 VMs/host	45	34.2	7.9
6 VMs/host	12	25.4	2.5
7 VMs/host	7	24.8	0.9

Table 2: **EC2 Disk Performance vs. VM co-location:** Write performance vs. number of VMs per physical host on EC2. Second column shows how many VMs fell into each load level.

5.1.1 Impact of Contention on I/O Performance

In the first test, we timed 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 machines exhibit a "cold start" phenomenon where the first write to a block is slower than subsequent writes, possibly to expand the VM's disk allocation, we "warmed up" 5000 MB of space on each machine before we ran our tests, by running `dd` and deleting its output.

We used a `traceroute` from each VM 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 7 VMs located on one physical host. Table 2 shows average performance and standard deviations. Performance ranged from 62 MB/s for the isolated VMs to 25 MB/s when seven VMs shared a host.

To validate that the performance was tied to contention for disk resources due to multiple VMs writing on the same host, we also tried performing `dd`'s in a smaller EC2 allocation where 200 VMs were assigned to 200 distinct 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 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 machine. From these results, we conclude that background load is an important factor in I/O performance on EC2, and can reduce I/O performance by a factor of 2.5. We also see that stragglers can occur "in the wild" on EC2.

We also measured I/O performance on "large" and "extra-large" EC2 VMs. These VMs have 2 and 4 virtual disks respectively, which appear to be independent. 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 on average less than that of small VMs.

5.1.2 Impact of Contention at the Application Level

We also evaluated the hypothesis that background load reduces the performance of Hadoop. For this purpose, we ran two tests with 100 virtual machines: one where each VM was on a separate physical host that was doing no other work, and one where all 100 VMs were packed onto 13 physical hosts, with 7 machines per host. These tests were in EC2's test cluster, where we had allocated all 800 VMs. 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 VMs, the job completed in 408s, whereas with VMs packed densely onto physical hosts, it took 1094s. Therefore there is a 2.7x difference in Hadoop performance with a cluster of isolated VMs versus a cluster of colocated VMs.

5.2 Scheduling Experiments on EC2

We evaluated LATE, Hadoop's native scheduler, and no speculation in a variety of experiments on EC2, on clusters of about 200 VMs. For each experiment in this section, we performed 5-7 runs. Due to the environment's variability, some of the results had high variance. To address this issue, we show the average, worst and best-case performance for LATE in our results. We also ran experiments on a smaller local cluster where we had full control over the environment for further validation.

We compared the three schedulers in two settings: Heterogeneous but non-faulty nodes, chosen by assigning a varying number of VMs to each physical host, and an environment with stragglers, created by running CPU and I/O intensive processes on some machines. We wanted to show that LATE provides gains in heterogeneous environments even if there are no faulty nodes.

As described at the start of Section 5, we ran these experiments in an EC2 test cluster where we allocated 800 VMs on 106 physical nodes – nearly the full capacity, since each physical machine seems to support at most 8 VMs – and we selected a subset of the VMs for each test to control colocation and hence contention.

5.2.1 Scheduling in a Heterogeneous Cluster

For our first experiment, we created a heterogeneous cluster by assigning different numbers of VMs to physical hosts. We used 1 to 7 VMs per host, for a total of 243 VMs, as shown in Table 3. 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 3: Load level mix in our heterogeneous EC2 cluster.

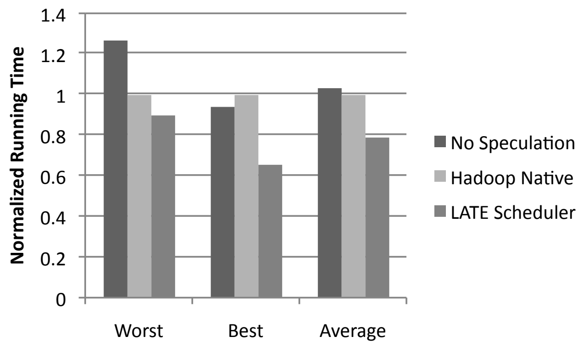

Figure 3: **EC2 Sort running times in heterogeneous cluster:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

As our workload, we used a Sort job on a data set of 128 MB per host, or 30 GB of total data. 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 shows the response time achieved by each scheduler. Our graphs throughout this 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, because the variance is high. On average, in this first experiment, LATE finished jobs 27% faster than Hadoop's native scheduler and 31% faster than no speculation.

5.2.2 Scheduling with Stragglers

To evaluate the speculative execution algorithms on the problem they were meant to address – faulty nodes – we manually slowed down eight VMs in a cluster of 100 with background processes to simulate stragglers. The other machines were assigned between 1 and 8 VMs per host, with about 10 in each load level. The stragglers

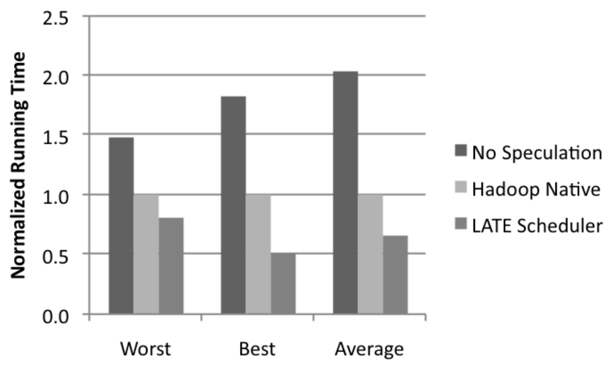

Figure 4: **EC2 Sort running times with stragglers:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

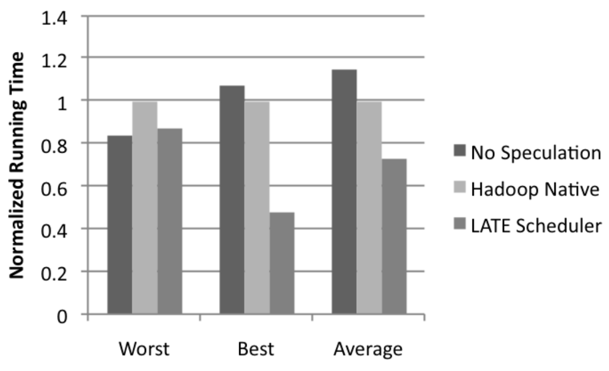

Figure 5: **EC2 Grep running times with stragglers:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

were created by running four CPU-intensive processes (tight loops modifying 800 KB arrays) and four disk-intensive processes (`dd` tasks creating large files in a loop) on each straggler. The load was significant enough that disabling speculative tasks caused the cluster to perform 2 to 4 times slower than it did with LATE, 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 4 shows the results of 4 experiments. On average, LATE finished jobs 58% faster than Hadoop's native scheduler and 220% faster than Hadoop with speculative execution disabled. The speed improvement over native speculative execution could be as high as 93%.

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 stragglers. These are example jobs that come with the Hadoop distribution. We used a 204-node cluster with 1 to 8 VMs per physical host. We simulated eight stragglers with background load 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 (less than a minute). 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". Results from 5 runs are shown in Figure 5. On average, LATE finished jobs 36% faster than Hadoop's native scheduler and 57% faster than no speculation.

We notice that in one of the experiments, LATE performed worse than no speculation. This is not surprising given the variance in the results. We also note that

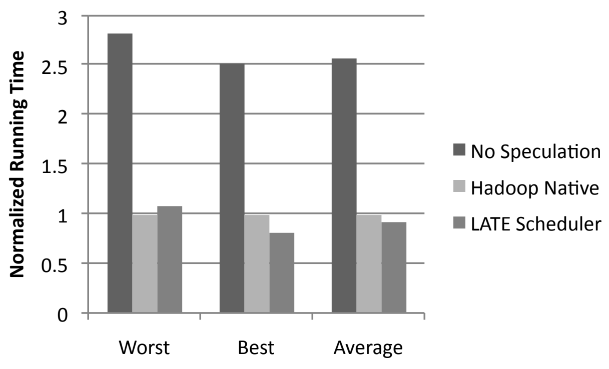

Figure 6: **EC2 WordCount running times with stragglers:** Worst, best and average-case performance of LATE against Hadoop's scheduler and no speculation.

there is an element of "luck" involved in these tests: if a data chunk's two replicas both happen to be placed on stragglers, then no scheduling algorithm can perform very well, because this chunk will be slow to serve.

WordCount counts the number of occurrences of each word in a file. We applied WordCount to a smaller data set of 21 GB, or 100 MB per host. Results from 5 runs are shown in Figure 6. 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 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. 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

Load Level	VMs	Write Perf (MB/s)	Std Dev
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 4: **Local cluster disk performance:** Write performance vs. VMs per host on local cluster. The second column shows how many VMs fell into each load level.

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

Table 5: Load level mix in our heterogeneous local cluster.

they will be equally slow with LATE and native speculation. In contrast, in jobs where reducers do more work, maps are a smaller fraction of the total time, and LATE has more opportunity to outperform Hadoop's scheduler. Nonetheless, speculation was helpful in all tests.

5.3 Local Testbed 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 [13].

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 computing resources to allow a large range of virtual machines per host (1-4).

5.3.1 Local I/O 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 to a file. We captured the timing of each `dd` command and show the averaged results of 10 runs in Table 4. We saw 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, whereas in the worst case on EC2, 8 VMs were contending for 4 disks.

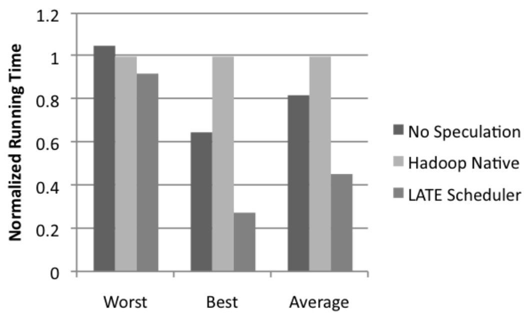

Figure 7: **Local Sort with heterogeneity:** Worst, best and average-case times for LATE against Hadoop's scheduler and no speculation.

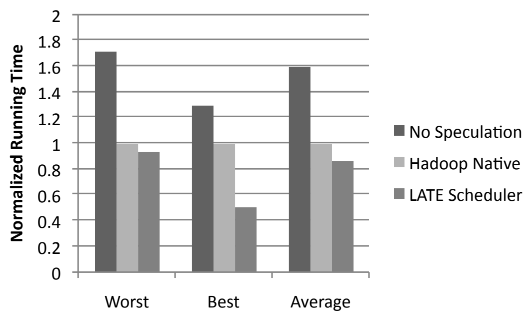

Figure 8: **Local Sort with stragglers:** Worst, best and average-case times for LATE against Hadoop's scheduler and no speculation.

5.3.2 Local Scheduling Experiments

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 scaled the allocation to the size of the hardware we were using, as shown in Table 5. We then ran the Hadoop Sort benchmark on 64 MB of input data per node, for 5 runs. Figure 7 shows the results. On average, LATE finished jobs 162% faster than Hadoop's native scheduler and 104% faster than no speculation. The gain over native speculation could be as high as 261%.

We also tested an environment with stragglers by running intensive background processes on two nodes. Figure 8 shows the results. On average, LATE finished jobs 53% faster than Hadoop's native scheduler and 121% faster than Hadoop with speculative execution disabled.

Finally, we also tested the WordCount workload in the local environment with stragglers. The results are shown in Figure 9. We see that LATE performs better on average than the competition, although as on EC2, the gain is less due to the nature of the workload.

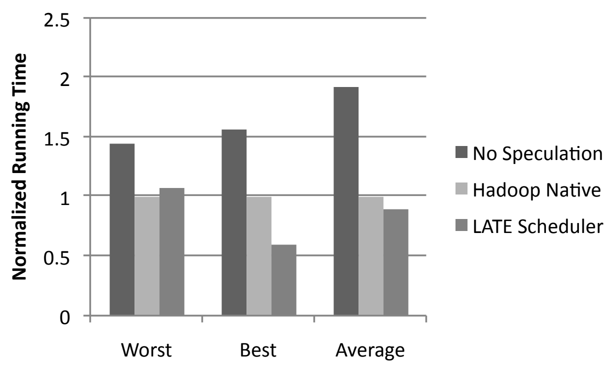

Figure 9: **Local WordCount with stragglers:** Worst, best and average-case times for LATE against Hadoop's scheduler and no speculation.

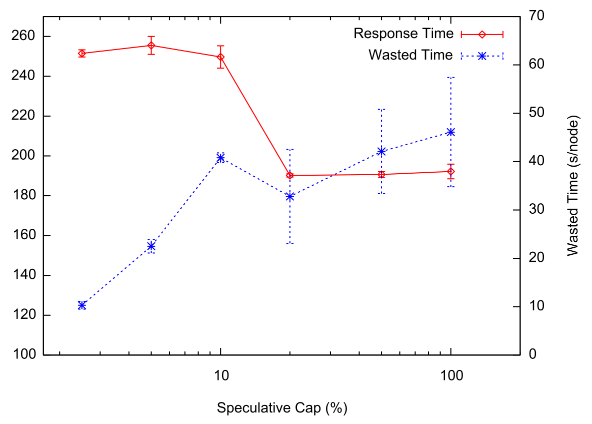

Figure 10: Performance versus SpeculativeCap.

5.4 Sensitivity Analysis

To verify that LATE is not overly sensitive to the thresholds defined in 4, we ran a sensitivity analysis comparing performance at different values for the thresholds. For this analysis, we chose to use a synthetic sleep workload, where each machine was deterministically "slowed down" by a different amount. The reason was that, by this point, we had been moved out of the test cluster in EC2 to the production environment, because the bug that initially put is in the test cluster was fixed. In the production environment, it was more difficult to slow down machines with background load, because we could not easily control VM colocation as we did in the test cluster. Furthermore, there was a risk of other users' traffic causing unpredictable load. To reduce variance and ensure that the experiment is reproducible, we chose to run a synthetic workload based on behavior observed in Sort.

Our job consisted of a fast 15-second map phase followed by a slower reduce phase, motivated by the fact that maps are much faster than reduces in the Sort job. Maps in Sort only read data and bucket it, while reduces merge and sort the output of multiple maps. Our job's reducers chose a sleep time t on each machine based on a per-machine "slowdown factor". They then slept 100 times for random periods between 0 and $2t$, leading to uneven but steady progress. The base sleep time was 0.7 seconds, for a total of 70s per reduce. We ran on 40 machines. The slowdown factors on most machines were 1 or 1.5 (to simulate small variations), but five machines had a sleep factor of 3, and one had a sleep factor of 10, simulating a faulty node.

One flaw in our sensitivity experiments is that the sleep workload does not penalize the scheduler for launching too many speculative tasks, because sleep tasks do not compete for disk and network bandwidth. Nonetheless, we chose this job to make results reproducible. To check that we are not launching too many speculative tasks, we measured the time spent in killed tasks in each test. We also compared LATE to Hadoop's native scheduler and saw that LATE wasted less time speculating while achieving faster response times.

5.4.1 Sensitivity to SpeculativeCap

We started by varying $SpeculativeCap$, that is, the percentage of slots that can be used for speculative tasks at any given time. We kept the other two thresholds, $SlowTaskThreshold$ and $SlowNodeThreshold$, at 25%, which was a well-performing value, as we shall see later. We ran experiments at six $SpeculativeCap$ values from 2.5% to 100%, repeating each one 5 times. Figure 10 shows the results, with error bars for standard deviations. We plot two metrics on this figure: the response time, and the amount of *wasted time* per node, which we define as the total compute time spent in tasks that will eventually be killed (either because they are overtaken by a speculative task, or because an original task finishes before its speculative copy).

We see that response time drops sharply at $SpeculativeCap = 20\%$, after which it stays low. Thus we postulate that any value of $SpeculativeCap$ beyond some minimum threshold needed to speculatively re-execute the severe stragglers will be adequate, as LATE will prioritize the slowest stragglers first. Of course, a higher threshold value is undesirable because LATE wastes more time on excess speculation. However, we see that the amount of wasted time does not grow rapidly, so there is a wide operating range. It is also interesting to note that at a *low* threshold of 10%, we have more wasted time than at 20%, because while fewer speculative tasks are launched, the job runs longer, so more *time* is wasted in tasks that eventually get killed.

As a sanity check, we also ran Hadoop with native

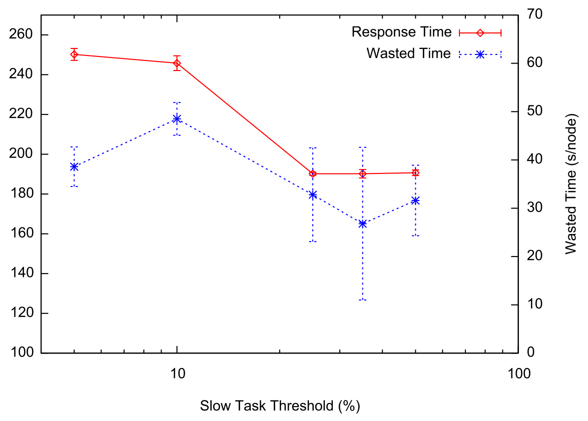

Figure 11: Performance versus SlowTaskThreshold.

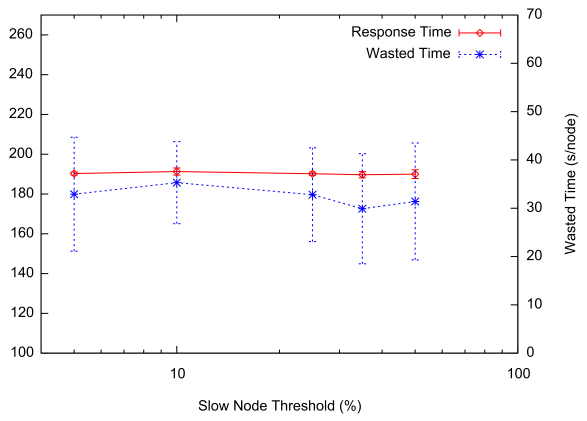

Figure 12: Performance versus SlowNodeThreshold.

speculation and with no speculation. Native speculation had a response time of 247s (std dev 22s), and wasted time of 35s/node (std dev 16s), both of which are worse than LATE with $SlowCapThreshold = 20\%$. No speculation had an average response time of 745s (about 10×70s, as expected) and, of course, 0 wasted time.

Finally, we note that the optimal value for $SpeculativeCap$ in these sensitivity experiments, 20%, was larger than the value we used in our evaluation on EC2, 10%. The 10% threshold probably performed poorly in the sensitivity experiment because 6 out of our 40 nodes, or about 15%, were slow (by 3x or 10x). Unfortunately, it was too late for us to re-run our EC2 test cluster experiments with other values of $SpeculativeCap$, because we no longer had access to the test cluster. Nonetheless, we believe that performance in those experiments could only have gotten better with a larger $SpeculativeCap$, because the sensitivity results presented here show that after some minimum threshold, response time stays low and wasted work does not increase greatly. It is also possible that there were few enough stragglers in the large-scale experiments that a 10% cap was already high enough.

5.4.2 Sensitivity to SlowTaskThreshold

$SlowTaskThreshold$ is the percentile of progress rate below which a task must lie to be considered for speculation (e.g. slowest 5%). The idea is to avoid wasted work by not speculating tasks that are progressing fast when they are the only non-speculated tasks left. For our tests varying this threshold, we set $SpeculativeCap$ to the best value from the previous experiment, 20%, and set $SlowNodeThreshold$ to 25%, a well-performing value. We tested 6 values of $SlowTaskThreshold$, from 5% to 100%. Figure 11 shows the results. We see again that while small threshold values harmfully limit the number of speculative tasks, values past 25% all work well.

5.4.3 Sensitivity to SlowNodeThreshold

$SlowNodeThreshold$ is the percentile of speed below which a node will be considered too slow for LATE to launch speculative tasks on. This value is important because it protects LATE against launching a speculative task on a node that is slow but happens to have a free slot when LATE needs to make a scheduling decision. Launching such a task is wasteful and also means that we cannot re-speculate that task on any other node, because we allow only one speculative copy of each task to run at any time. Unfortunately, our sleep workload did not present any cases in which $SlowNodeThreshold$ was necessary for good performance. This happened because the slowest nodes (3x and 10x slower than the rest) were so slow that they never finished their tasks by the time the job completed, so they were never considered for running speculative tasks.

Nonetheless, it is easy to construct scenarios where $SlowNodeThreshold$ makes a difference. For example, suppose we had 10 fast nodes that could run tasks in 1 minute, one node X that takes 2.9 minutes to run a task, and one node Y that takes 10 minutes to ran a task. Supposed we launched a job with 32 tasks. During the first two minutes, each of the fast nodes would run 2 tasks and be assigned a third. Then, at time 2.9, node X would finish, and there would be no non-speculative task to give it, so it would be assigned a speculative copy of the task on node Y. The job as a whole would finish at time 5.8, when X finishes this speculative task. In contrast, if we waited 0.1 more minutes and assigned the speculative task to a fast node, we would finish at time 4, which is 45% faster. This is why we included $SlowNodeThreshold$ in our algorithm. As long as the threshold is high enough that the very slow nodes fall below it, LATE will make reasonable decisions. Therefore we ran our evaluation with a value of 25%, expecting that fewer than 25% of nodes in a realistic environment will be severe stragglers.

For completeness, Figure 12 shows the results of varying $SlowNodeThreshold$ from 5% to 50% while fixing $SpeculativeCap = 20\%$ and $SlowTaskThreshold = 25\%$. As noted, the threshold has no significant effect on performance. However, it is comforting to see that the very high threshold of 50% did not lead to a *decrease* in performance by unnecessarily limiting the set of nodes we can run speculative tasks on. This further supports the argument that, as long as $SlowNodeThreshold$ is higher than the fraction of nodes that are extremely slow or faulty, LATE performs well.

6 Discussion

Our work is motivated by two trends: increased interest from a variety of organizations in large-scale data-intensive computing, spurred by decreased storage costs and availability of open-source frameworks like Hadoop, and the advent of virtualized data centers, exemplified by Amazon's Elastic Compute Cloud. We believe that both trends will continue. Now that MapReduce has become a well-known technique, more and more organizations are starting to use it. For example, Facebook, a relatively young web company, has built a 300-node data warehouse using Hadoop in the past two years [14]. Many other companies and research groups are also using Hadoop [5, 6]. EC2 also growing in popularity. It powers a number of startup companies, and it has enabled established organizations to rent capacity by the hour for running large computations [17]. Utility computing is also attractive to researchers, because it enables them to run scalability experiments without having to own large numbers of machines, as we did in our paper. Services like EC2 also level the playing field between research institutions by reducing infrastructure costs. Finally, even without utility computing motivating our work, heterogeneity will be a problem in private data centers as multiple generations of hardware accumulate and virtualization starts being used for management and consolidation. These factors mean that dealing with stragglers in MapReduce-like workloads will be an increasingly important problem.

Although selecting speculative tasks initially seems like a simple problem, we have shown that it is surprisingly subtle. First, simple thresholds, such as Hadoop's 20% progress rule, can fail in spectacular ways (see Section 3.2) when there is more heterogeneity than expected. Other work on identifying slow tasks, such as [15], suggests using the mean and the variance of the progress rate to set a threshold, which seems like a more reasonable approach. However, even here there is a problem: identifying *slow* tasks *eventually* is not enough. What matters is identifying *the tasks that will hurt response time the most*, and doing so *as early as possible*. Identifying a task as a laggard when it has run for more than two standard deviations than the mean is not very helpful for reducing response time: by this time, the job could have already run 3x longer than it should have! For this reason, LATE is based on estimated time left, and can detect the slow task early on. A few other elements, such as a cap on speculative tasks, ensure reasonable behavior. Through our experience with Hadoop, we have gained substantial insight into the implications of heterogeneity on distributed applications. We take away four lessons:

1. **Make decisions early,** rather than waiting to base decisions on measurements of mean and variance.
2. **Use finishing times,** not progress rates, to prioritize among tasks to speculate.
3. **Nodes are not equal.** Avoid assigning speculative tasks to slow nodes.
4. **Resources are precious.** Caps should be used to guard against overloading the system.

7 Related Work

MapReduce was described architecturally and evaluated for end-to-end performance in [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 problem of speculative execution, motivated by the challenges we observed in heterogeneous environments.

Much work has been done on the problem of scheduling policies for task assignment to hosts in distributed systems [18, 19]. However, this previous work deals with scheduling independent tasks among a set of servers, such as web servers answering HTTP requests. The goal is to achieve good response time for the average *task*, and the challenge is that task sizes may be heterogeneous. In contrast, our work deals with improving response time for a job consisting of *multiple* tasks, and our challenge is that node speeds may be heterogeneous.

Our work is also related to multiprocessor task scheduling with processor heterogeneity [20] and with task duplication when using dependency graphs [21]. Our work differs significantly from this literature because we focus on an environment where node speeds are unknown and vary over time, and where tasks are shared-nothing. Multiprocessor task scheduling work focuses on environments where processor speeds, although heterogeneous, are known in advance, and tasks are highly interdependent due to intertask communication. This means that, in the multiprocessor setting, it is both possible and necessary to plan task assignments in advance, whereas in MapReduce, the scheduler must react dynamically to conditions in the environment.

Speculative execution in MapReduce shares some ideas with "speculative execution" in distributed file systems [11], configuration management [22], and information gathering [23]. However, while this literature is focused on guessing along decision branches, LATE focuses on guessing which running tasks can be overtaken to reduce the response time of a distributed computation.

Finally, DataSynapse, Inc. holds a patent which details speculative execution for scheduling in a distributed computing platform [15]. The patent proposes using mean speed, normalized mean, standard deviation, and fraction of waiting versus pending tasks associated with each active job to detect slow tasks. However, as discussed in Section 6, detecting slow tasks *eventually* is not sufficient for a good response time. LATE identifies the tasks that will hurt response time the most, and does so as early as possible, rather than waiting until a mean and standard deviation can be computed with confidence.

8 Conclusion

Motivated by the real-world problem of node heterogeneity, we have analyzed the problem of speculative execution in MapReduce. We identified flaws with both the particular threshold-based scheduling algorithm in Hadoop and with progress-rate-based algorithms in general. We designed a simple, robust scheduling algorithm, LATE, which uses estimated finish times to speculatively execute the tasks that hurt the response time the most. LATE performs significantly better than Hadoop's default speculative execution algorithm in real workloads on Amazon's Elastic Compute Cloud.

9 Acknowledgments

We would like to thank Michael Armbrust, Rodrigo Fonseca, George Porter, David Patterson, Armando Fox, and our industrial contacts at Yahoo!, Facebook and Amazon for their help with this work. We are also grateful to our shepherd, Marvin Theimer, for his feedback. 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, VMWare, Siemens, Amazon and Facebook.

References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Communications of the ACM, 51 (1): 107-113, 2008.

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

[3] Amazon Elastic Compute Cloud, `http://aws.amazon.com/ec2`

[4] Yahoo! Launches World's Largest Hadoop Production Application, `http://tinyurl.com/2hgzv7`

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

[6] Presentations by S. Schlosser and J. Lin at the 2008 Hadoop Summit. `tinyurl.com/4a6lza`

[7] D. Gottfrid, Self-service, Prorated Super Computing Fun, New York Times Blog, `tinyurl.com/2pjh5n`

[8] Figure from slide deck on MapReduce from Google academic cluster, `tinyurl.com/4zl6f5`. Available under Creative Commons Attribution 2.5 License.

[9] R. Pike, S. Dorward, R. Griesemer, S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall, Scientific Programming Journal, 13 (4): 227-298, Oct. 2005.

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

[11] E.B. Nightingale, P.M. Chen, and J.Flinn. Speculative execution in a distributed file system. ACM Trans. Comput. Syst., 24 (4): 361-392, November 2006.

[12] Amazon EC2 Instance Types, `tinyurl.com/3zjlrd`

[13] 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. ACM SOSP 2003.

[14] Personal communication with the Yahoo! Hadoop team and with Joydeep Sen Sarma from Facebook.

[15] J. Bernardin, P. Lee, J. 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.

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

[17] EC2 Case Studies, `tinyurl.com/46vyut`

[18] Mor Harchol-Balter, Task Assignment with Unknown Duration. Journal of the ACM, 49 (2): 260-288, 2002.

[19] 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, pp. 268-269, 1998.

[20] B.Ucar, C.Aykanat, K.Kaya, and M.Ikinci. Task assignment in heterogeneous computing systems. J. of Parallel and Distributed Computing, 66 (1): 32-46, Jan 2006.

[21] S.Manoharan. Effect of task duplication on the assignment of dependency graphs. Parallel Comput., 27 (3): 257-268, 2001.

[22] Y. Su, M. Attariyan, J. Flinn AutoBash: improving configuration management with operating system causality analysis. ACM SOSP 2007.

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