Managing Tiny Tasks for Data-Parallel, Subsampling Workloads

Sundeep Kambhampati*, Jaimie Kelley*, Christopher Stewart*, William C.L. Stewart‡, Rajiv Ramnath§
* Computer Science and Engineering, The Ohio State University
‡The Research Institute, Nationwide Children's Hospital
‡Department of Pediatrics & Statistics, The Ohio State University
§CERCS for Enterprise Transformation and Innovation, The Ohio State University

***Abstract*—Subsampling workloads compute statistics from a set of observed samples using a random subset of sample data (i.e., a subsample). Data-parallel platforms group these samples into tasks; each task subsamples its data in parallel. In this paper, we study subsampling workloads that benefit from tiny tasks—i.e., tasks comprising few samples. Tiny tasks reduce processor cache misses caused by random subsampling, which speeds up per-task running time. However, they can also cause significant scheduling overheads that negate the time reduction from reduced cache misses. For example, vanilla Hadoop takes longer to start tiny tasks than to run them. We compared the task scheduling overheads of vanilla Hadoop, lightweight Hadoop setups, and BashReduce. BashReduce, the best platform, outperformed the worst by 3.6X but scheduling overhead was still 12% of a task's running time. We improved BashReduce's scheduler by allowing it to size tasks according to kneepoints on the miss rate curve. We tested these changes on high-throughput genotype data and on data obtained from Netflix. Our improved BashReduce outperformed vanilla Hadoop by almost 3X and completed short, interactive jobs almost as efficiently as long jobs. These results held at scale and across diverse, heterogeneous hardware.**

I. Introduction

Internet services and mobile devices have generated large amounts of data. Indeed, 90% of all data has been produced in the last two years [7]. Data will continue to grow as other types of data collection become popular. For example, genome sequencing has become 1,000X cheaper over the last 5 years [25]. As costs continue to decrease, genomic data alone could produce an exabyte of data. Processing this data, especially for interactive workloads, is challenging. Subsampling is a statistical approach that computes means, modes, and percentiles using only randomly selected portions of each data sample. As an example, consider a family that participates in an study on Bi-Polar Disorder. The family's genetic data is a sample that comprises many AT/CG base pairs. A subsampling workload may examine randomly selected base pairs to determine whether the family line shares a certain gene. Subsampling trades accuracy for speed, enabling interactive, big-data workloads while allowing for some statistical error.

Subsampling workloads can run on data-parallel platforms, e.g., Hadoop, in map-reduce jobs. These platforms scale out by partitioning sampled data across multiple nodes. Each node subsamples within map tasks, producing intermediate results from randomly selected data. Reduce tasks combine these intermediate results. However, subsampling workloads differ from traditional Hadoop workloads because the map tasks access randomly selected portions of data. These random accesses can cause L2 cache misses, forcing processors to fetch data from main memory. For tasks that would otherwise achieve low cache miss rates, random access patterns can significantly increase cache miss rates, degrading processing efficiency.

Our key insight is that subsampling workloads benefit from *tiny tasks*, i.e., map tasks that randomly sample from only a small portion of the sampled data stored on a node. Although data-parallel platforms must process more tiny tasks for the same result, random accesses within tiny tasks are less likely to cause cache misses. Tiny tasks complete efficiently, wasting few CPU cycles on retrieving data from main memory (or disk). However, tiny tasks present scheduling challenges for data-parallel platforms. First, platforms must start tiny tasks efficiently or increased startup costs will negate efficiency gains. Second, platforms must improve runtime efficiency to avoid slowing down quickly completing tiny tasks.

For this paper, we set up a data-parallel platform that supports tiny tasks and speeds up subsampling. We used a two step approach. First, we benchmarked existing platforms in terms of tiny-task scheduling overhead. We compared three Hadoop configurations and BashReduce (a lightweight implementation of the map-reduce paradigm [10]). Vanilla Hadoop took approximately 4X longer to start tasks compared to BashReduce. A second version of Hadoop, in which we disabled task level recovery and speculative execution, had reduced overheads, and a third version, in which no HDFS data transfer occurred, achieved very low overheads. Second, we implemented a new task sizing approach for the BashReduce scheduler. Our approach sizes tasks to the first kneepoint on an empirical task size to miss rate curve. By doing so, we lower the scheduling overhead for tiny tasks.

We set up two subsampling workloads. EAGLET finds disease genes from subsamples of dense SNP linkage data within the DNA of sampled families [34]. Our Netflix workloads describe customer rating patterns by subsampling user ratings of sampled movies. With low overhead and tiny-task sizing, our BashReduce platform sped up EAGLET and Netflix workloads by 3X and 2.5X compared to vanilla Hadoop. We achieved 25% speedup compared to a lightweight Hadoop setup that had low overhead but no task sizing. Our platform achieved 12X speedup on small input sizes where whole jobs complete within minutes, making our platform attractive for workloads governed by service level objectives [27], [32], [42].

On the EAGLET workload, our platform achieved 117 Mb/s per 12-core node, comparing favorably against competing map-reduce platforms for secondary genetic analysis [30], [31]. Throughput scaled linearly as we allocated additional resources. Our platform also scaled linearly within virtualized environments. In a heterogeneous environment, our platform was limited by the last task to finish its work. For small jobs, throughput degraded proportionately to the slowest task to complete. For larger jobs, however, tiny tasks facilitated workload stealing, erasing slowdowns [2], [39], [41].

Our Contributions: This paper focuses on interactive, data-parallel workloads [16], [21], [26], [27], [30], [32]. Map and reduce tasks within these workloads complete quickly, relying on efficient processing and on low scheduling overhead [27]. Our contributions include:

1. We make the case for tiny tasks in subsampling workloads, by quantifying cache miss rates as task size increases.
2. We measure scheduling overheads on tiny tasks, i.e., startup and runtime costs, in existing data-parallel platforms.
3. We implemented a task sizing algorithm within the BashReduce scheduler to reduce runtime overheads.
4. We experimentally validate our improved BashReduce platform, comparing it to vanilla and lightweight Hadoop setups across multiple workloads and diverse clusters.

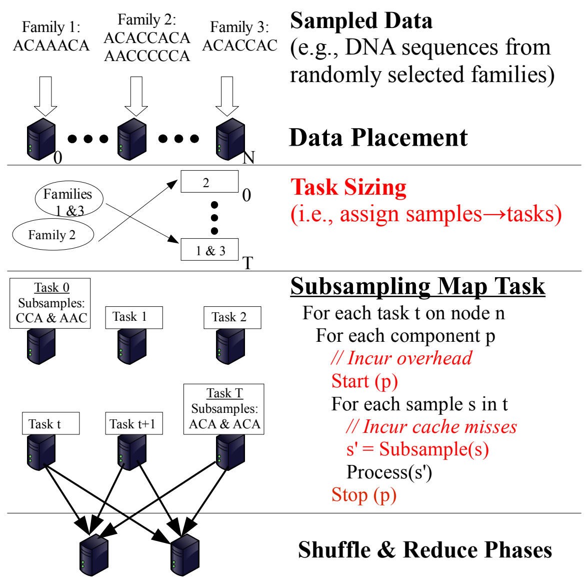
In the remainder of this paper, Section II describes subsampling, task size, and their effect on cache locality. Section III benchmarks per-task overheads in widely used data-parallel platforms. These overheads led us to integrate task sizing within the BashReduce scheduler. Experiments in Sections IV and V show that our improved BashReduce achieves high throughput and responsiveness. Section VI discusses related work and Section VII concludes.

II. Subsampling Workloads

Figure 1 depicts and labels stages for data-parallel subsampling. For these workloads, input data is grouped by some feature (e.g., by family id). Each unit of grouped data is called a sample. Normally, the space of potential samples is much larger than the number of observed samples. Sample and subsample sizes vary as depicted in Figure 1.

Data-parallel platforms place data samples across many nodes; these nodes then process the data in parallel. When nodes access data stored remotely, parallel processing slows down. Data placement affects performance greatly. In the best case, a copy of each sample is stored on each node, eliminating remote accesses. However, such full replication is only feasible for small datasets. In practice, each sample is stored on only a few nodes and some nodes store more samples than others. Such data skew will cause remote data accesses when nodes with few samples try to steal work from heavily loaded nodes [2]. Load balancing and handling data skew were the focus of [2], [39]. Our research is orthogonal to this research.

A task comprises the software components used to process samples (p in Figure 1). A task's size is the number of samples processed by each component invocation. A task size of S_n starts each component only once, using that invocation to process all samples and piping all results between components. Here, S_n is the number of samples on node n. If the task size is set close to S_n, we call the resulting task a *large task*. Large tasks avoid scheduling delays caused by cloning processes, managing temporary files, and context switching. However, subsampling workloads present a challenge: Access patterns within subsampling software are random. Large tasks that process many subsamples can exhibit poor locality on their input dataset.



```
For each task t on node n
  For each component p
    // Incur overhead
    Start (p)
    For each sample s in t
      // Incur cache misses
      s' = Subsample(s)
      Process(s')
    Stop (p)
```

Fig. 1: Data flow for a data-parallel subsampling workload.

On the other extreme, a task size of 1 starts and stops each software component for each sample. We define *tiny tasks* as tasks with size close to 1. Tiny tasks suffer from scheduling delays but their region for random data access is much smaller. After compulsory cache misses, tiny tasks often exhibit good cache locality.

This paper focuses on task sizing for data-parallel workloads that must complete within seconds or minutes. Platforms that support these workloads increasingly store data within main memory, ensuring data access delays are low. Examples of such platforms include Pig [42], RDD [40], Data Cube [24], Sparrow [27], and [16]. These workloads may support interactive analysis of scientific data, personalized advertising, sentiment analysis, or real-time trace studies [42]. Whether tasks are large or small, each task produces intermediate results that are forwarded to the shuffle and reduce stage. Interactive workloads often have relatively short reduce phases. If the reduce stage consumes a large fraction of a workload's execution time, task sizing for an efficient map stage has low impact [41].

A. The Case for Tiny Tasks

In traditional data-parallel workloads, programmers know which data locations will be accessed during a map task. Their software components preload this data in fast processor caches, speeding up data access by orders of magnitude. However, in non-traditional data parallel workloads, which use subsampling to access only a fraction of available data per task, programmers do not know exactly which data samples will be accessed. By definition, subsampling tasks must randomly

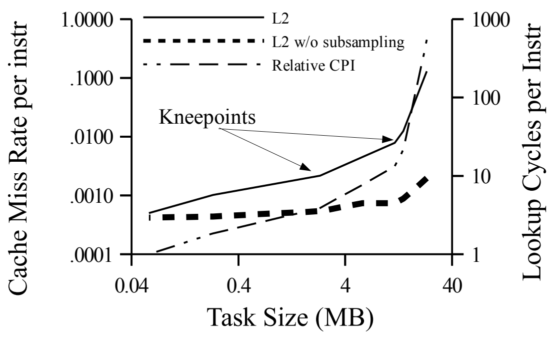


Fig. 2: L2 misses per instruction and cycles per instruction across task sizes in EAGLET.

choose which data subsamples to use during runtime. As task size grows, these random accesses are likely to cause misses in processor caches. A large task processes more samples than a tiny task, incurring more cache misses.

Figure 2 makes the case for tiny tasks on the EAGLET subsampling workload. EAGLET (Efficient Analysis of Genetic Linkage: Testing and Estimation) finds genomic sequences correlated with diseases [34]. Samples in EAGLET reflect DNA from families (i.e., grandparents, parents, and children) that volunteered to be sequenced. EAGLET accepts a list of family IDs as input. It outputs intermediate, weighted statistical data. Intermediate data can be combined to produce statistics across the entire dataset. We started with 230 MB of real data consisting of 400 samples from a linkage study on bi-polar disorder and scaled it as needed. In practice, scientists use EAGLET as the first step to detect disease genes. Before requesting costly lab work to confirm their hypothesis, scientists may use EAGLET to test up to 10^5 genomic sequences for statistical correlations. EAGLET jobs should complete each test as quickly as possible to allow scientists to interactively refine their hypotheses.

In Figure 2, the task size presented in MB reflects the number of families included in EAGLET's input list. At runtime, EAGLET randomly selects subsets of each family's genome, looks for the genomic marker, and computes intermediate results. Intermediate results from different tasks are combined during the reduce phase. These functions are divided across multiple widely used, open-source software components, including MERLIN, Perl, GenLib, and others.

We used OProfile [19] to sample cache misses while EAGLET ran. We set up Oprofile to distinguish EAGLET's subsampling program from other programs. We ran these experiments on an Intel Sandy Bridge processor with 6 dual cores with 1.5MB L2 cache and 15MB L3 cache. We observed that large tasks incurred higher miss rates. A 25MB-sized task saw 35X more L2 cache misses per instruction than a 2.5MB-sized task. The EAGLET subsampling component is the source of the increase missed rate. The miss rate was flat among other components.

Random accesses increase the cache miss rate in two ways. First, the data being accessed is unlikely to be in cache, causing compulsory misses. Second, they represent unique data accesses that evict other, potential useful data from LRU caches [12]. Stack distance is the number of unique data references between accesses to the same data. Stack distances smaller than the cache size means data accesses will hit in cache. Random accesses (due to subsampling) injected between normal accesses make cache hits less likely. This explains a key property of Figure 2: *The miss rate changes at certain key task-size thresholds.* After those points, increasing the task size results in random accesses evicting frequently accessed data that normally, i.e., without subsampling, would have hit in cache. We call points where the miss rate increased sharply kneepoints. Kneepoints were at 2.5MB and 11MB. Separately, we also captured cache misses in the L3 caches and observed a kneepoint at 11MB.

Cache misses force tasks to retrieve data from memory. On the Intel Sandy Bridge, data access from memory is 63X slower than L2 cache hits. Average memory access time *(AMAT)* per instruction, the time for a lookup in the fastest cache plus the product of the miss rate and the miss penalty, is a well-known model to study the effect of cache misses [28]. The secondary axis on Figure 2 plots the normalized AMAT where the fastest cache looks up results in 1 cycle. We observed over a 1,000X increase in AMAT between the tiniest task and the largest task.

III. Managing Tiny Tasks

Tiny tasks have fewer cache misses per instruction than large tasks. However, data-parallel platforms configured to use tiny tasks will start and stop software components more often than platforms configured to use large tasks. The time taken to schedule software components, called *scheduling overhead*, may exceed the time saved by improved cache locality.

Hadoop monitors each task's execution for potential node or disk failures. On failure, tasks are restarted with different resources. The monitoring and data replication required for such task-level recovery are major sources of scheduling overhead. Job-level recovery, in which a node or disk failure would restart the whole job, can lower scheduling overhead [30]. In this section, we first make a case for job-level recovery in interactive data-parallel workloads. Then, we quantify scheduling overhead in data parallel platforms, comparing a vanilla Hadoop setup, lightweight Hadoop setups, and a clean-slate platform. We reduce scheduling overhead by moving toward job-level recovery.

A. *Job- vs Task-level Recovery*

Hadoop was designed to process multiple petabytes spread over $10^4 - 10^5$ nodes [38], taking hours or days to complete a map-reduce job. During the course of a job execution, multiple disks and nodes were likely to fail. If each failure restarted the entire workload, the job would never complete on Hadoop, making the decision for task-level recovery on Hadoop simple.

We revisit task-level recovery here in the context of interactive, subsampling workloads that run for minutes. The shorter time frame makes it $10^3 - 10^4$ times less likely that a failure will occur in the midst of a job execution. Further, these workloads use fewer nodes because 1) data stored in main memory is costly [16], [27], [40] and 2) their goal is often

Codename	Core	Task-level Failures	Full Dist. File Sys.	Java
Vanilla Hadoop	Hadoop	Yes	Yes	Yes
Job-level Hadoop	Hadoop	No	Yes	Yes
Lite Hadoop	Hadoop	No	No	Yes
BashReduce [10]	Unix Utilities	No	No	No

TABLE I: Platforms benchmarked for this paper

to compute results from iterative or incremental changes [21], [22].

Mechanisms for task-level recovery, e.g., monitoring and data replication, increase a workload's running time. Let $cost_{tl}$ be the slowdown factor. On failures, only tasks are restarted, rather than entire jobs. On each failure, task-level recovery saves the difference between the expected job and task running times. Our key insight is that task-level fault tolerance only makes sense if 1) hardware failures occur faster than jobs complete, meaning every job is likely to see a failure or 2) rerunning entire jobs would slow down running time by more than $cost_{tl}$. For short, interactive workloads, the latter concern is most important.

Let $mttf$ represent the mean time to a node or disk failure. Also, let $\bar{P}(w)$ reflect of service level objective (SLO) for the workload [42]—i.e., the worst case running time. We expect at most ($f_w = N \cdot \frac{\bar{P}(w)}{mttf} \cdot \alpha$) failures during an execution. Here, α captures correlated, heavy-tail failures that occur within the SLO window. We now compute f_w for typical subsampling workloads. We set $\bar{P}(w) = 10$ minutes and $\alpha = 1.5$. Taking guidance from recent work [16], [27], [41], we set $N = 100$. We set $mttf = 4.3$ months from [13], [30]. Under these settings, $f_w = 0.0078$, meaning that monitoring overhead would have to fall below 1% to justify task-level recovery. Next, we quantify actual overheads observed in Hadoop.

B. Platform Selection

We measured scheduling overhead for the platforms shown in Table I. Here, we describe the salient features of each platform. More details are can be found in Section IV.

Hadoop was an obvious choice to benchmark, as it is widely used in practice for map reduce workloads. *Vanilla Hadoop* used default monitoring and HDFS policies. Each task reports its progress to a central service that exposes an HTTP front end. Also, tasks use HDFS instead of the local Linux file system. In the job-level Hadoop setup, we disabled the central monitoring service. In the lite Hadoop setup, we modified EAGLET so that map tasks created no intermediate HDFS files, avoiding replication costs. This new version of EAGLET performed calculations based on a static, globally distributed file rather a dynamic file. We also disabled the central monitoring service in lite Hadoop. Note, lite Hadoop is shown for benchmarking only—its results are incorrect.

The BashReduce platform takes a clean-slate approach [10]. It is a very lightweight implementation of the map reduce paradigm based on running tasks within the Bash shell. These tasks are connected through simple TCP pipes using the *nc6* tool. Task-level fault tolerance has never been supported in

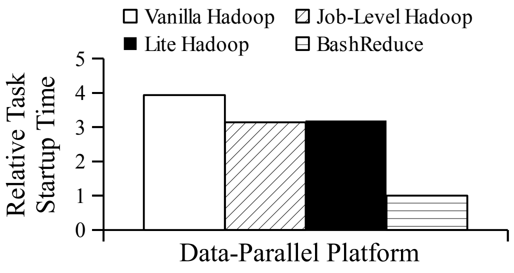


Fig. 3: Relative time to start 1 task on each core by platform

BashReduce. BashReduce also elides a global distributed file system (HDFS). Managers partition data and tasks access only the local file system. In future work, we include comparisons for Sparrow [27] and Piccolo [29].

We quantified two types of scheduling overhead. *Startup time* captures delays that happen only once for each workload. These delays include TCP handshakes for longstanding connections and data staging. *Runtime overhead* captures delays incurred as a task runs. Specifically, runtime overhead is the difference in running time between running software components directly on Linux and running them on one of the platforms in Table I. We ran these experiments on a 72-core cluster consisting of 6 dual-core Intel Sandy Bridge processors. Each core served as a map slot. Task size was fixed at 1 sample.

We measured startup time by running a hello-world job where tasks equaled map slots. Each task was identical and completed within milliseconds (less than 0.01% of the job's running time on Hadoop). Figure 3 shows the time taken to complete this job. Times are normalized to the overhead of BashReduce. Task monitoring overhead increased Hadoop's startup costs by 21%, about 52 seconds. Task-level failures would have to recover hundreds of sub-second subsampling tasks to justify this large overhead. Using formulas from the previous section, clusters smaller than 30K nodes do not justify 21% overhead. BashReduce could start jobs almost 4X faster than vanilla Hadoop.

Figure 4 compares the relative per-task runtime overhead of each platform. For this test, we ran an EAGLET subsampling workload comprised of 4K tasks and measured the total running time. Then we subtracted the startup time and divided by 4K. The result is shown relative to the running time on Linux without a platform. Failure monitoring caused a 20% degradation per task. However, the largest runtime gain came from bypassing HDFS on short-lived temporary files. Indeed, the experiment on Linux without a platform achieved runtime overhead almost equal to BashReduce's overhead. BashReduce still incurred 12% overhead due to scheduling the subsampling map tasks on the cluster. In practice, this overhead would accumulate for tiny tasks. In the next section, we address this overhead by looking for relatively large tiny tasks.

C. Task Sizing

Per-task scheduling overhead penalizes many tiny tasks more than few large tasks. Large tasks amortize per-task delays, e.g., creating Linux processes, across many samples.

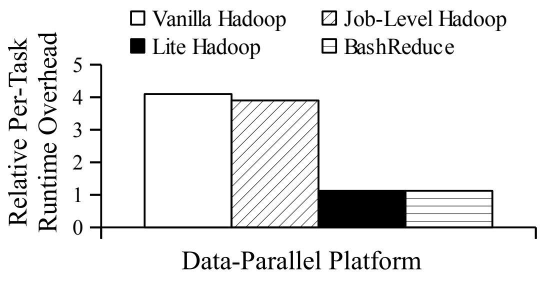


Fig. 4: Runtime overhead of each platform relative to native Linux

Offline: Determine Kneepoint

```
public static int kneepoint(int maxSampleNum) {
  float[2] taskSizes = new float[];
  float[2] missRates = new float[];

  //Pick random samples for study
  float[] samples = RandomArray(1, maxSampleNum);
  List workingSet = new List();
  workingSet.add(samples[0]);

  // Run the tiniest task and collect misses
  results = ExecTask(workingSet);
  misses[0] = results.cacheMisses();
  taskSizes[0] = results.inputSize();

  int growthRate = 0, i = 0;
  float MAX_RATE = -1;
  // Run tests at each size, compare miss rates
  while ((growthRate <= MAX_RATE) ||
        (MAX_RATE == -1)) {
    workingSet.add(samples[i]);
    results = ExecTask(workingSet);
    missRates[1] = results.cacheMisses();
    taskSizes[1] = results.inputSize();
    growthRate = ((missRates[1] – missRates[0]))
                     /((taskSizes[1] – taskSizes[0]));
  //bookeeping
    if (MAX_RATE == -1) MAX_RATE = growthRate;
    missRates[0] = missRates[1];
    taskSizes[0] = taskSizes[1];
    i++;
  }
  return (taskSize(i-1));
}
```

Runtime Scheduler: Task Sizing

```
public void sizing(int kneepoint,
                 InputStream dataset) {
  // determine size in terms of # samples
  float AVG_SAMPLE_SIZE = K;

  int size = kneepoint / AVG_SAMPLE_SIZE;

  //Split dataset into tasks
  InputStream[] tasks;
  tasks = splitInputStream(dataset, size);
  for(InputStream task: tasks){
    addToMapJobList(task);
  }
  // start Bash Reduce
  StartBashReduce();
}
```

Fig. 5: Java code for offline kneepoint detection and task sizing implemented within BashReduce.

However, very large tasks face large cache miss rates. In this subsection, we present a task-sizing approach. *We size tasks at the smallest kneepoint on the task size to miss rate curve (i.e., Figure 2)*. The smallest kneepoint is the largest task size before the first increase in the cache-miss growth rate. Our approach achieves low cache miss rates while amortizing per-task overhead across samples. We implemented task sizing within the BashReduce scheduler. In an offline step, we created the task size to miss rate curve and found kneepoints. In an online step, we packed subsamples into tasks.

Specifically, Figure 5 outlines our approach. First, during an offline phase, we collect data on the relationship between task size and cache misses. On a benchmarking node, we run Oprofile. We run map tasks in isolation, varying the number of samples in the task's working set. As seen in Figure 2, we plot the aggregate input data size against cache misses per instruction. We modified our BashReduce platform to group samples into tasks of equal (kneepoint) size before starting map tasks. We place the same number of samples in each task, assuming samples are roughly the same size; in practice, data parallel jobs have large outliers [3], [16]. Our genetic analysis dataset also has outliers, with one sample 15X larger than the mean and a second sample 7X larger than the mean. The time taken by the offline phase is about 3% of the time taken by the online phase. However, the offline phase is a one time overhead paid for each new data set. In future work, we are developing a dynamic task sizing approach that can adapt to outliers rigorously and reduce the overhead of offline computation.

We compared the impact of task sizing on BashReduce's performance. We ran EAGLET on the 72-core Sandy Bridge cluster. EAGLET subsampled data and computed genetic statistics 30 times for each family. Each of these subsamples (i.e., 30 x 400 families) could run in its own map slot. Figure 6 shows throughput relative to 24MB large tasks, i.e., the amount of data partitioned to each map slot in the cluster (S_n). Our results include the delay for determining the kneepoint offline.

First, we removed outlier samples from our dataset (shown as no outliers in Figure 6). Outlier samples run 50X longer compared to the mean run time, or longer. We observed that our kneepoint approach achieved 15% speedup compared to the baseline created by the 24MB large task approach. Further, the tiniest task approach caused 8% slowdown. When we included the outlier samples, we observed that our approach increased throughput by 23%. This is because the outlier tasks increased the cache miss rate within their task groups by pushing valuable data out of the cache. Tiny tasks were more helpful under the heterogeneous workload. The absolute running time with hetergeneous tasks under the tiniest task approach was 791 seconds with outliers, and 322 seconds without the outliers. Outliers themselves caused a 2.4X slow down [3], [32]. Our task sizing approach had a larger impact with outliers but did not overcome the slow down caused by outliers.

Discussion: The kneepoints identified by our offline analysis are contingent on hardware and workload. The task size to miss rate curve should be recomputed if processor cache sizes or data access patterns change. Our ongoing work attempts to identify a cross-platform heuristic to identify kneepoints, especially for cloud platforms where processor cache sizes

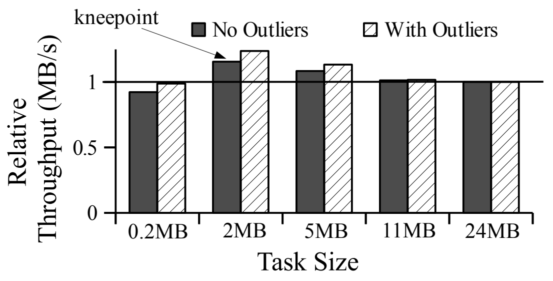


Fig. 6: Impact of our our kneepoint algoirithm on runtime

are not known. Our experiments in the next section show that kneepoint selection is insensitive to small errors.

IV. Experimental Setup

We set up two subsampling workloads. EAGLET [34], described earlier, is open-source software that finds disease-causing genes from a collection of sequenced families. Our dataset originally comprised DNA sequences of 400 families (over 4,000 individuals) who volunteered for a Bi-Polar study, but we grew this data as needed. The data of a single family is represented in a data sample from this workload. The workload recomputed analysis that unveiled well-known linkages [4]. In total, the original data exceeded 230 MB. As is common practice in genetic analysis, we ran the workload 30 times for each sample, making the job size 6.9 GB (i.e., 230 MB x 30). For larger tests, we created synthetic data based on patterns in the original data. Our largest test was a 1 TB job spanning 684K families. The distribution of family sizes (and hence sample sizes) was heavy tailed. Outliers were preserved in our synthetic data.

We also set up a subsampling workload based on Netflix movie ratings [1], [41]. Here, each sample represents a movie that Netflix streamed to its users. The data within each sample are tuples composed of the date, user id, and the user's rating of the movie. Our workload subsampled ratings for each movie to estimate typical user ratings by month. Data size was 2 GB with 118 KB per movie. By subsampling, we found the user ratings faster than exhaustive calculation would have [41] but we also allowed errors to occur. We classified two types of Netflix workloads: High confidence and low confidence. The high confidence workload estimates average user ratings with a 98% confidence interval, choosing less speedup and more accuracy. The low confidence workload estimates use two orders of magnitude fewer ratings, accepting more error for speedup.

Task Sizing: Our EAGLET and Netflix workloads differed in terms of software complexity. EAGLET used multiple (> 5) open-source software packages that spanned three programming languages. Our Netflix workloads used only Bash scripts. We hypothesized that EAGLET was more likely to suffer from tiny-task scheduling overhead.

Both workloads used a pointer to a file containing the actual input data. If the file was large and contained many samples, the task operating on the file was large. If the file was small and contained few samples, the resulting task was tiny. Precisely, we define large tasks as jobs that consist of all of the samples partitioned to a node (i.e., S_n samples in 1 file). The tiniest tasks have S_n files that are piped one-by-one into the respective programs.

Platforms: We compared the following platforms.

1. BashReduce w/ Task Sizing (BTS): We set up BashReduce [10] with *netcat* for inter-node communication via pipes. BashReduce centralizes scheduling and shuffling stages on a single *master node.* In our setup, the master node also decides on task sizes by creating input files locally and distributing them to all other *worker nodes*. The master node includes the offline script described in Figure 5. Unless otherwise mentioned, BTS sets task size to 2.5 MB for EAGLET and 1 MB for Netflix. If any master or worker node fails, the entire BashReduce job is restarted.

2. BashReduce w/ Large Tasks (BLT): In this setup, the master node referred to all samples on a node within a single file.

3. BashReduce w/ Tiniest Tasks (BTT): In this setup, the master node referred to only 1 sample in each of S_n input files.

4. Vanilla Hadoop (VH): We compared other platforms against Hadoop, a widely used platform for data analysis. Our default configurations uses an HDFS replication factor of $\frac{N}{2}$ to reduce data migration traffic. A large replication factor is a sensible optimization for interactive workloads that use relatively small datasets. Each node is configured to have as many map slots as cores.

5. Job-Level Hadoop (JLH) disables TaskTracker, the feature responsible for task level recovery. Also, speculative execution is disabled. These optimizations make Hadoop more suitable for our interactive workloads by reducing task startup and runtime overheads.

6. Lite Hadoop (LH): This benchmark produces incorrect results but achieves very low overhead on the Hadoop platform. We use it to benchmark overhead from Java Runtime and to understand the potential for revised subsampling-aware Hadoop. We changed EAGLET so that it fixes intermediate files used to pass data between software components. The subsampling portion of EAGLET was unaffected. We set the replication factor to N on the intermediate files, ensuring no HDFS data transfer would slow down the platform.

Hardware: We used a private cloud with three types of servers, shown in Table II. Processors include AMD and Intel brands that vary by cache size, memory capacity, and processing speed. Our experimental setup restricted the amount of hardware available to focus on performance improvement using limited hardware.

V. Experimental Results

Figure 7 compares the BashReduce setups. For this test, we used 6 nodes of hardware type 1 (See Table II). In total, the tests ran on 72 cores. These tests used only the original data from the Bi-Polar study and movie ratings. We observed that BTS achieved throughput 10–90% higher than BLT and 26–32% higher than BTS. Because the Netflix sampling workload uses fewer software components than EAGLET, it was able

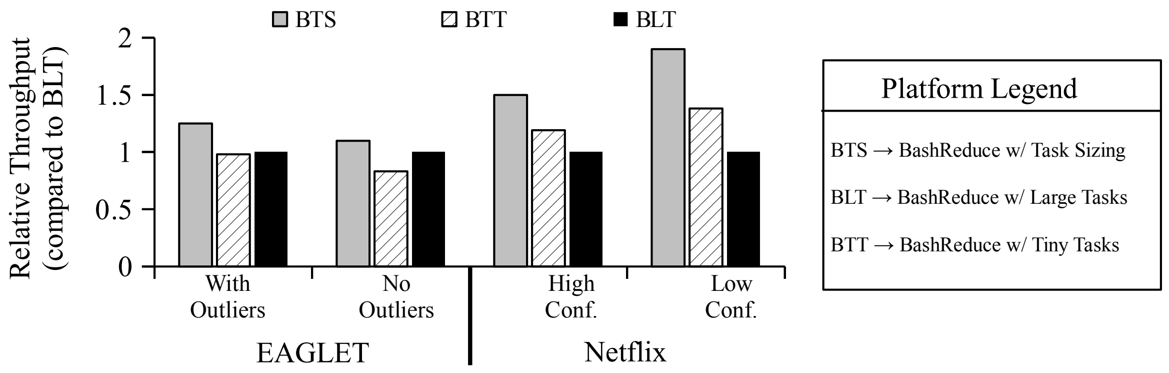


Fig. 7: BTS speeds up both EAGLET and Netflix workloads relative to BLT and BTT. Tests ran on hardware type 1. The rightmost table provides acronyms for all of the platforms referenced in this section.

	Type 1	Type 2	Type 3
Processor	Xeon	Xeon	Opteron
Cores per Node	12	12	32
Processing Speed	2.0G	2.3G	2.3G
L2 Cache	15MB	15MB	32MB
Memory	32GB	32GB	64GB
Virtualized	No	No	Yes

TABLE II: Hardware used in our studies.

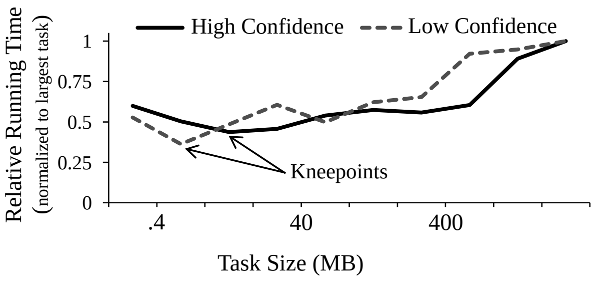


Fig. 8: Kneepoints in the Netflix subsampling workloads on BashReduce.

to better exploit cache locality, resulting in favorable BTT results. In contrast, EAGLET suffered additional per-task runtime overhead from starting many software components on tiny tasks. BTS balances these issues, typically outperforming its closest competitor by 17%.

Figure 8 shows that kneepoints occurred for the Netflix workloads as well. Results shown were run on top of BashReduce. However, the kneepoints occurred at different places for the high and low confidence workloads despite subsampling the same data. We expected this result because cache locality patterns varied depending on the confidence level desired. Our offline approach can find a different kneepoint depending on the workload, provided the data is available. For results presented in this section, we used only 1 kneepoint (1 MB) for both Netflix workloads. Results with high confidence workload in Figure 7 show that exact kneepoints are not needed to improve throughput relative to BLT and BTT. To quantify how robust our approach is, we created five Netflix workloads that varied according to their output confidence level. Among the five workloads, the 1 MB task size ranked in the top 2 task sizes (in terms of throughput) three times. In the cases where it was not the best performing task size, it was within 10% of the best. Further, the 1 MB task size setting outperformed large and tiniest task settings in all 5 workloads.

BTS versus Hadoop: Hadoop is a widely used platform for data processing. However, it is not designed for short, interactive jobs [38]. We compared the throughput of BTS to three Hadoop setups across different job sizes. For these tests, we ran the EAGLET subsampling workload on type 2 hardware, varying job size. We changed the job size by adding synthetic families to the Bi-Polar data.

Figure 9 shows that BTS sped up VH by almost 5X on jobs with a 12 MB task size. For reference, we found that a 12 MB job can test a genetic hypothesis on 40 families with 15 subsamples per family. As the job size increased, BTS offered less speedup because VH was able to amortize its startup costs. We recall here that JLH had lower startup costs and runtime overhead compared to VH. JLH performs better on short jobs, but BTS still offered 3.7X speedup.

Along with tracking task-level failures, the Hadoop platform monitors CPU utilization, I/O efficiency and other system metrics. The metrics are queried frequently to produce user-friendly web displays about the state of the system. We added system level monitoring into BTS. We used Oprofile [19] to capture L2 and L3 cache misses, instruction counts, accesses to memory, and CPU utilization data. We collected this data every second, sending it to a central node for display. We do not claim that our approach rivals the sophistication of Hadoop (i.e., production code). Instead, our goal was to understand the impact of adding monitoring on BTS. We observed that BTS with monitoring suffered a 21% slowdown on MB-sized jobs, due to the increased startup overhead. On GB-sized jobs or larger, the runtime overhead caused an additional 15% slowdown. Despite these delays, BTS with monitoring still speeds up JLH by 2.5X on small jobs and 1.5X on larger jobs.

EAGLET allows scientists to test genetic hypotheses before sending them away for costly lab work. This process could proceed much faster if it were interactive. Before this work, we observed that vanilla EAGLET (i.e., without Hadoop or

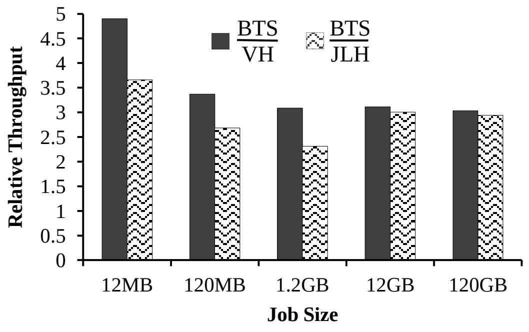


Fig. 9: Comparison of BTS to VH and JLH.

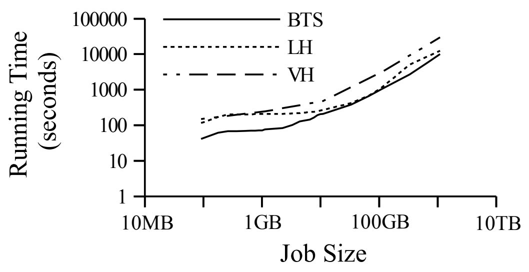


Fig. 10: Comparison of BTS to VH and LH in terms of running time. Note log-log scale.

BashReduce) took an hour to complete a 230 MB job on a type 2 node; it was not designed for parallel execution. Running EAGLET within Hadoop and BashReduce platforms improved performance by using all available cores. Figure 10 shows BTS's speedup over VH. These tests used 72 cores of type 2 hardware. With BTS, we completed a 91 MB job in 40 seconds. The same job took 150 seconds to run on VH. A 230 MB job ran on BTS in 68 seconds, a 59X speedup over vanilla EAGLET on 12 cores. For comparison to the state of the art, recent studies with CloudBlast, a competing tool for secondary genetic analysis, achieved 60 Mb/s [30] and 24 Mb/s [31]. BTS sustains 117 Mb/s. Note, these results are anecdotal. We can not compare them directly because the workloads differ.

We also compared against LH. LH suffered from high startup costs when job sizes were small, essentially matching VH up to 1.1 GB sized jobs. It never achieved response times within 100 seconds. As job size increased, LH approached BTS performance. However, BTS (due to task scheduling) maintained 25% throughput gain even under a 1 TB job size.

Elasticity: Figure 11 shows throughput as we changed the number of cores in BTS. The platform scaled linearly up to 1 TB job. These tests were conducted on a 1 Gb/S network. The 72-core test (i.e., 6 type 2 nodes) produced results at 45% of network capacity. In Figure 11, regions where 72-core throughput equalled 36-core performance reflected startup costs. Large job sizes amortize these costs. For interactive workloads that run small jobs, however, the 72-core tests wasted resources. Managers should scale out until additional cores provide diminishing returns and no further.

Service-level objectives guarantee that a job will finish within a fixed running time [6], [32], [41], [42]. For data processing workloads, a job's running time depends on its

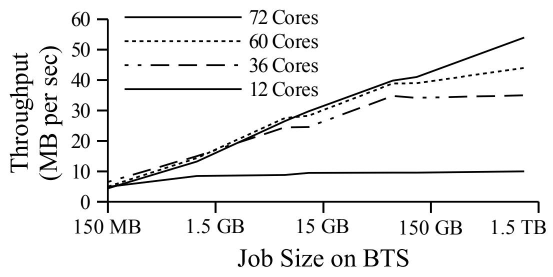


Fig. 11: EAGLET on BTS as number of cores changed.

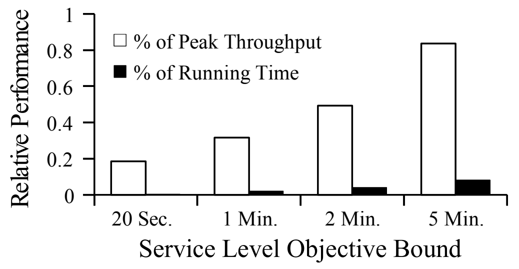
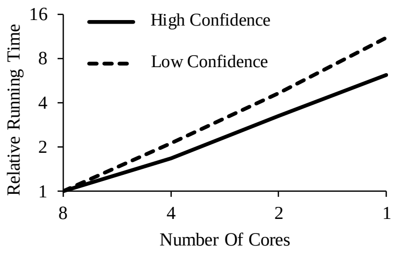


Fig. 12: The throughput and running time of EAGLET on BTS clusters scaled to efficiently meet service level objectives.

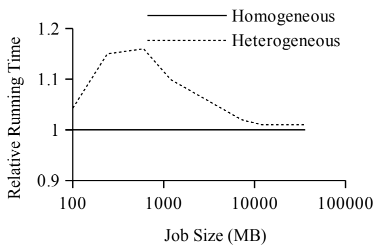
size and the platform's achieved throughput at that size. If the job size is too small, startup costs dominate, limiting the data that can be processed within the fixed running time. Figure 12 shows BTS performance under various service level objectives. Each result reflects the platform configuration with highest achieved throughput within the fixed running time. Note, the 72-core case was only the best for 2-minute and 5-minute bounds. It has high startup costs, which allows the 36-core and 12-core case to perform better under tight bounds. Figure 12 shows performance relative to BTS's peak throughput without any service level objective. For reference, we also show the fixed running time relative to the running time when peak throughput was achieved. We observed that under a 2 minute SLO BTS achieved 50% of its peak throughput. For reference, a 2 minute SLO represents 4% of the 50 minute run time needed to achieve peak throughput on 72 cores. A 5 minute SLO achieved 83% of peak throughput.

Virtualization and Heterogeneity: We tested our workloads on user-mode Linux virtual machines. For these tests, we used the original datasets for each workload. Each virtual machine was allocated 1 AMD Opteron core (i.e., type 3 in Table II). We re-ran our task sizing algorithm on this hardware; EAGLET had a kneepoint at a task size of 1.2 MB and Netflix had a kneepoint at a task size of 1 MB. Compared to type 2 hardware, i.e., without virtualization, we observed slowdown of 16% across both workloads. BTS still scaled out well, Figure 13(a) shows linear improvement for the Netflix workload.

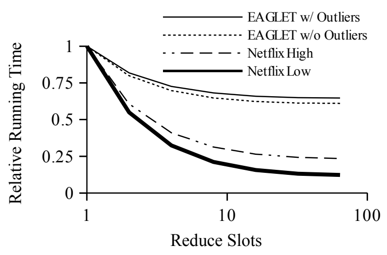
We tested BTS under a heterogeneous environments where 12 of 60 cores were 15% slower than the others (i.e., 1 slow node). The slow node was of type 1 hardware and the others were of type 3 hardware. The slow nodes caused proportional slowdown on MB-sized jobs. However, as job size grew, BTS's round robin scheduler skipped over busy, slower



(a) Netflix workload as cores scale on Type 3.



(b) Netflix workload as job size increases.



(c) Network resource demand increases.

Fig. 13: Running time on BTS

cores, assigning more tasks to the faster cores. As a result, the performance loss is divided across 48 cores.

Finally, we studied the impact of reduce tasks. The BashReduce platform does not support multiple reduce slots gracefully. It requires mapping data back to all nodes and running the reduce stage as a map stage in an interactive computation. We used simulation to understand the impact of multiple reduce stages, and corresponding communication delay. We used formulas from [41] to understand the expected performance as reduce tasks increase. We calibrated these models with average map time, reduce time, and shuffle time from our experiments with 1-node map reduce. Figure 13(c) highlights the results. With EAGLET, secondary genetic analysis is compute intensive [30]. As a result, adding reduce tasks quickly exhibits diminishing returns. The Netflix workload, however, can speed up at the reduce stage.

VI. Related Work

It is challenging to coordinate processors, routers, memory controllers, and disks in parallel, especially for interactive workloads. In this section, we describe recent papers on scheduling algorithms, data storage architectures, modeling approaches, and workload-specific designs. These papers advanced the state of the art for interactive, data-parallel platforms. In comparison, this paper targets subsampling, data-parallel workloads. We show that task size affects data access times and design a platform and scheduler to support tiny tasks.

Large clusters provide resources shared by many data platforms. These platforms have their own schedulers that may independently and accidentally overload nodes, causing transient queuing delays. As we observed, even seemingly small delays have large effects on tiny tasks. The Sparrow scheduler [26], [27] presents a data-parallel version of power-of-two load balancing [23] that allows independent schedulers to avoid transient delays. Each node's operating system and background jobs also cause transient delays. Replication for predictability [3], [16], [32] sends requests to multiple nodes and takes the first response, masking transient delays. Within local networks, individual paths can become overloaded. These issues are hard to resolve because application and network interactions are opaque [9]. Mizan [18] focuses on Pregel workloads, providing a high throughput scheduler that balances network I/O between vertex queues.

Moore's law proves that exploiting parallelism offers diminishing returns for execution time. Platforms should use enough parallel resources to achieve service level objectives but no more. Zhang et al. [42] model execution time for Pig, a platform for iterative map-reduce, as a function of parallel resources used. Such *performance models* can be used to make online management decisions [33]. GreenHadoop and GreenSlot [14], [15] also create accurate performance models. However, their focus was exploiting intermittent renewable energy for reduced carbon footprint [11]. AMAT (average memory access time) is a simple model that makes a strong point: faster storage can significantly decrease execution times. RDD [40], Data Cube [24], Pig [42],FCS [36] and [16] lower execution times by using main memory for storage, rather than disk. However, main memory is volatile and costly. Often, it is paired with disk or SSD in hybrid storage. Tsai et al. [35] provide a framework to compare caching and partitioned hybrid architectures. hStorageDB is one such hybrid system [20].

Graph workloads often run tasks starting from the same vertex multiple times. Each run differs because weights or edges from the vertex have changed slightly. These workloads can reduce their execution time by reusing results from prior tasks. Data mining and machine learning workloads have similar properties. McSherry et al. [21] propose language support for differential dataflow, a paradigm that allows programmers to specify incremental structure in their programs. RDD [40] users can call functions on cache misses, allowing for certain types of incremental workloads. Waterland et al. [37] cache results for parallel applications transparently within the operating system. Non-determinism presents a challenge for the above approaches. For example, results for our subsampling workloads are not easily cached by input data alone. One solution would cache random-seed keys along with data, but this may disturb the statistical power of subsampling. Other recent work has studied the efficiency of cloud caches, especially for data-parallel workloads [5], [8], [17].

VII. Conclusion

Many workloads now consist of more data than data-parallel platforms can process within interactive response time constraints. Subsampling reduces processing requirements while providing statistical confidence on the accuracy of results. In this paper, we studied subsampling workloads, showing that subsampling from a large working set can significantly degrade cache locality. We made a case for tiny tasks, i.e., splitting subsampling workloads into many tasks with small working sets. Tiny tasks offer improved cache locality but suffer from scheduling overheads. We contend that scheduling overheads can be managed. First, different platforms exhibit very different scheduling overheads depending on their objectives. Platforms designed for task-level recovery have overheads that are too high for tiny tasks. Platforms designed for job-level recovery perform better. Second, we show that task sizing can amortize some scheduling overheads with only a small increase in cache miss rate. Our approach uses kneepoints on the task size to miss rate curve to determine task size. We demonstrated the benefit of our approach using genetic analysis and e-commerce datasets. On short, interactive workloads, our improved platform performed 9X better than vanilla Hadoop.

VIII. Acknowledgements

This work was supported by NSF grant CNS 1320071. A portion of the work was done during Kambhampati's internship at Nationwide Children's Hospital.

References

[1] Netflix prize. http://www.netflixprize.com//index.

[2] F. Ahmad, S. Chakradhat, A. Raghunathan, and T. Vijaykumar. Tarazu: Optimizing MapReduce on Heterogeneous Clusters. In *ACM ASPLOS*, 2012.

[3] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *USENIX Symp. on Operating Systems Design and Implementation*, 2010.

[4] J. Badner and E. Gershon. Meta-analysis of whole-genome linkage scans of bipolar disorder and schizophrenia. *Molecular psychiatry*, 7(4):405–411, 2002.

[5] H. Bjornsson, G. Chockler, T. Saemundsson, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Tech Report*, 2013.

[6] S. Bouchenak. Automated control for sla-aware elastic clouds. In *Workshop on Feedback Computing*, 2010.

[7] L. Bradshaw. Big data and what it means. U.S. Chamber of Commerce Foundation, 2013.

[8] G. Chockler, G. Laden, and Y. Vigfusson. Design and implementation of caching services in the cloud. In *IBM Technical Report*, 2012.

[9] P. Costa. Bridging the gap between applications and networks in data centers. In *Sigops*, 2013.

[10] R. Crowley. BashReduce - Crowley Code. https://github.com/erikfrey/bashreduce.

[11] N. Deng, C. Stewart, J. Kelley, D. Gmach, and M. Arlitt. Adaptive green hosting. In *International Conference on Autonomic Computing*, 2012.

[12] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Notices*, volume 38, 2003.

[13] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, , and S. Quinlan. Availability in globally distributed storage systems. In *USENIX Symp. on Operating Systems Design and Implementation*, 2010.

[14] I. n. Goiri, R. Beauchea, K. Le, T. D. Nguyen, M. E. Haque, J. Guitart, J. Torres, and R. Bianchini. Greenslot: scheduling energy consumption in green datacenters. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[15] I. n. Goiri, K. Le, T. D. Nguyen, J. Guitart, J. Torres, and R. Bianchini. Greenhadoop: leveraging green energy in data-processing frameworks. In *European Conference on Computer Systems*, 2012.

[16] J. Kelley and C. Stewart. Balanced and predictable networked storage. In *International Workshop on Data Center Performance*, 2013.

[17] J. Kelley, C. Stewart, Y. He, and S. Elnikety. Cache provisioning for interactive NLP services. In *Workshop on Large and Distributed Systems (LADIS)*, 2013.

[18] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *European Conference on Computer Systems*, 2013.

[19] J. Levon and P. Elie. OProfile - A System Profiler for Linux. http://oprofile.sourceforge.net/.

[20] T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang. hstorage-db: heterogeneity-aware data management to exploit full capacity of hybrid storage systems. In *VLDB*, 2012.

[21] F. McSherry, R. Isaacs, M. Isard, and D. Murray. Differential dataflow. In *CIDR*, 2013.

[22] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, 2010.

[23] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 2001.

[24] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. 2011.

[25] National Human Genome Research Institute. Dna sequencing costs. http://www.genome.gov/sequencingcosts/, 2013.

[26] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *HotOS*, 2013.

[27] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Scalable scheduling for sub-second parallel jobs. In *ACM Symp. on Operating Systems Principles*, 2013.

[28] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[29] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *USENIX Symp. on Operating Systems Design and Implementation*, 2010.

[30] A. Rasmussen, M. Conley, R. Kapoor, V. Lam, G. Porter, and A. Vahdat. Themis: An i/o efficient mapreduce. In *ACM Symp. on Cloud Computing*, Oct. 2012.

[31] M. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. In *BioInformatics*, 2009.

[32] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *International Conference on Autonomic Computing*, 2013.

[33] C. Stewart, K. Shen, A. Iyengar, and J. Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2010.

[34] W. C. Stewart, E. N. Drill, D. A. Greenberg, et al. Finding disease genes: a fast and flexible approach for analyzing high-throughput data. *European Journal of Human Genetics*, 19(10):1090, 2011.

[35] C. Tsai, J. Chou, and Y. Chung. Value-based tiering management on heterogeneous block-level storage system. In *CloudCom*, 2012.

[36] Y. Wang, J. Tan, W. Yu, L. Zhang, and X. Meng. Preemptive reducetask scheduling for fair and fast job completion. In *Int'l Conf. on Autonomic Computing*.

[37] A. Waterland, J. Appavoo, and M. Seltzer. Parallelization by simulated tunneling. In *Workshop on Hot Topics in Parallelism*, 2012.

[38] T. White. Hadoop: The definitive guide. O'Reilly Media.

[39] J. Xie. Improving mapreduce performance in heterogeneous hadoop clusters. In *Intn'l Heterogeneity in Computing Workshop*, 2010.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Symp. on Networked Systems Design and Implementation*, 2012.

[41] Z. Zhang, L. Cherkasova, and B. Loo. Performance modeling of mapreduce jobs in heterogeneous cloud environments. In *IEEE CLOUD*, 2013.

[42] Z. Zhang, L. Cherkasova, A. Verma, and B. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Int'l Conf. on Autonomic Computing*, 2012.