Resource Allocation using Adaptive Characterization of Online, Data-Intensive Workloads

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Jaimie Kelley, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2017

Dissertation Committee:

Dr. Christopher Stewart, Advisor

Dr. Srinivasan Parthasarathy

Dr. P. Sadayappan

Abstract

Cloud resource providers balance maximizing utilization under a power cap with meeting workload Service Level Agreements (SLA). As the amount of data used by workloads increases, so do the pressures on compute capacity in the cloud. Even if the resources assigned meet an interactive workload's need for low latency, the data that interactive workload processes with allocated resources may not be sufficient to achieve a standard of answer quality. Increasing the resources allocated to a specific workload to meet its answer quality standard reduces the overall profit a cloud provider can make on interactive workloads. However, if a workload's answer quality standard is not met, the interactive workload may seek another placement. Cloud instances can be purchased by the minute, and multiple opportunities for placement exist. Because of this, cloud providers need to put their clients' interests first or lose revenue.

To best serve their own and their clients interest, cloud providers need data which reflects resource usage, answer quality, and service level. If a cloud provider knows the amount of power used by each workload scheduled, it can better fulfill its power cap requirements without penalty. If a cloud provider knows the current latency and answer quality of scheduled workloads, it can decide when to reallocate resources. However, this is difficult because any collection of data online imposes overheads. While cloud providers generally

reserve some percentage (5%) of utilization for operating system functions, data collection and analysis must be done carefully to avoid undue impact on scheduled workloads.

I use adaptive solutions to trade accuracy for overhead in workload characterization. Adaptive workload characterizations inform resource management without the high overhead of complete calculation, but are not completely accurate.

In my work, I adaptively reduce the time spent profiling peak power to the degree of accuracy that a cloud provider is willing to accept. I developed a model for adaptively profiling peak power usage to determine core scaling. Adaptive profiling saved up to 93% collection time while reducing accuracy by 3% on average.

To obtain answer quality for online resource management, I overlap execution of online requests with the execution of requests that use all relevant data by using memoization of complete responses from specific components. I built Ubora to obtain and allow management of answer quality for interactive, data-intensive workloads. Cloud providers set the rate at which queries are sampled, which exchanges overhead for accuracy.

Finally, I designed Quikolo, a service that speculatively deploys and characterizes a target workload in-situ in a colocation placement. Clients use this characterization to decide whether to migrate their workload to this available placement. Quikolo also enables study of overhead and accuracy influenced by the number of features and collection time used for workload characterization.

Adaptively trading accuracy reduces the impact of workload characterization on overhead. My adaptive characterization solutions enable cloud providers to provision for lower overhead and still achieve information that aids balancing client needs with available cloud resources.

Vita

2001-2005 ..... Lutheran High School West - Rocky River, OH

2005-2009 ..... Bachelor of Science, Computer Science and English (Writing), Heidelberg University

2010-2015 ..... Masters of Science, Computer Science and Engineering, The Ohio State University

2010-present ..... PhD student in Computer Science and Engineering, The Ohio State University

Publications

Jaimie Kelley, Christopher Stewart, Nathaniel Morris, Devesh Tiwari, Yuxiong He, and Sameh Elnikety, "Obtaining and Managing Answer Quality for Online Data-Intensive Services". *Journal ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 2017.

Jaimie Kelley, Christopher Stewart, Devesh Tiwari, and Saurabh Gupta, "Adaptive Power Profiling for Many-Core HPC Architectures". *International Conference on Autonomic Computing*, 2016.

Jaimie Kelley, Christopher Stewart, Devesh Tiwari, Sameh Elnikety, and Yuxiong He, "Measuring and Managing Answer Quality for Online Data-Intensive Services". *International Conference on Autonomic Computing*, 2015.

Sundeep Kambhampati, **Jaimie Kelley**, William C. L. Stewart, Christopher Stewart, and Rajiv Ramnath, "Managing Tiny Tasks for Data-Parallel, Subsampling Workloads". *IEEE International Conference on Cloud Engineering*, 2014.

Jaimie Kelley, Christopher Stewart, Sameh Elnikety, and Yuxiong He, "Cache Provisioning for Interactive NLP Services". *Large and Distributed Systems and Middleware*, 2013.

Jaimie Kelley and Christopher Stewart, "Balanced and Predictable Networked Storage". *International Workshop on Data Center Performance*, 2013.

Nan Deng, Christopher Stewart, **Jaimie Kelley**, Daniel Gmach and Martin Arlitt, "Adaptive Green Hosting". *International Conference on Autonomic Computing*, 2012.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Allocating cloud resources wisely is important because overprovisioning wastes potential revenue but underprovisioning loses customers. Cloud resource allocation is challenging because many customers must be satisfied with limited resources. Data centers operate under a global power cap, incurring a penalty if their power is over the limit by even a tenth of a second. This limits the number of cores that can be powered at any given point in time. Increasing utilization on a server is an alternative to increasing the number of powered cores, but carries penalties if workloads do not meet the expected latency guaranteed in their Service Level Agreements (SLA).

The problem of resource allocation in the cloud is exacerbated by the increasing amount of data used by workloads. In 2011, 1.8 zettabytes of data were created [42]. Data production increases by 40% each year [158]. This growth introduces problems in processing efficiency. 66% of data created in 2013 was generated by individuals, but corporations will be responsible for 85% of this data during some portion of its lifetime [158]. The growth of big data in all its facets will only continue in the coming years, with a projected size of 44 zettabytes worldwide by 2020 [158].

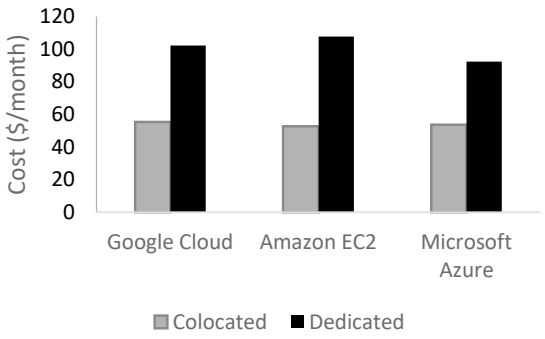

Figure 1.1: 4 cores in the cloud cost less if workloads colocate.

If not enough data is processed before a response is required, even workloads that are meeting their SLA might not be achieving satisfactory answers to requests. These workloads with low answer quality require more resources, without which their owners decide to seek new placement within a different cloud. This increase for a single workload reduces the amount of resources available to service new customers. Keeping current customers happy avoids losing this revenue.

Unhappy customers have multiple opportunities for alternative cloud placement. Google Cloud Platform, Microsoft Azure, and Amazon EC2 all sell competitive colocation and dedicated instances, as shown in Figure 1.1. Migrating to another cloud placement will lower a workload's operating cost when its SLA violations increase from resource pressure [111]. Therefore, it is in the cloud resource allocator's best interest to ensure each workload gets just enough resources.

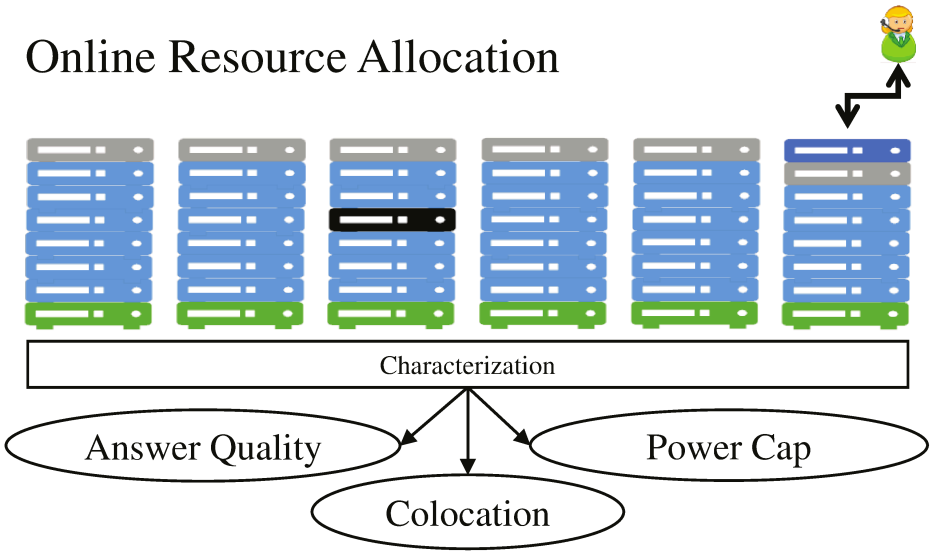

Figure 1.2: Clients connect to services in the cloud. I use workload characterization to solve resource allocation problems, including power capping, answer quality, and colocation assessment.

In order to ensure each workload gets just enough resources, cloud providers need data regarding that workload's latency, answer quality, and interactions with the cloud environment. For instance, the ratio of performance to power used by a workload can inform how many cores it should be allocated. Finding this knowledge improves a cloud provider's ability to meet its overall power cap requirements. Tracking request latency per workload warns the cloud provider when an SLA violation is imminent. Potential actions include allocating more resources, sprinting [104] or migrating the workload. The problem space is shown in Figure 1.2.

Characterizing and profiling are used synonymously within this work, but this is not generally the case in practice. To characterize a workload is to know the resources it uses, the arrival rate and service rate, or the number of users and query mix, depending on the type of characterization [101, 100]. To profile a workload is to extrapolate information regarding that workload, such as the peak power usage expected over the course of its execution [74]. To eliminate confusion, I used the term characterization when referring to all my work, but it is not always clear when one term should be used over the other. For instance, in Chapter 7, I continuously use the term characterization, but the end goal of this characterization is to use the gathered information to predict future tail latencies of the speculative workload execution, which is an extrapolation of information. Rather than switching between terms fluidly, I use profiling to refer to offline collection and analysis and characterizing to refer to online collection and analysis.

Unfortunately, any collection of data online incurs overhead. For applications like tracking online sports data, or error feeds at Facebook, data is stored and processed online as it arises [1]. Cloud providers typically reserve some percentage of utilization per machine for operating system functions and overhead [71]. More overhead than this carries the risk of increased SLA violations for workloads located on that machine.

In my research, I work with online, data-intensive workloads. Online, data-intensive services run on cloud resources, and have clients who expect fast response times (latency < 20 seconds). These services must process large quantities of data in parallel (e.g., 4 TB split across 32 threads). An example of such a service is found in Figure 1.3. Cloud infrastructures also host workloads which are expected to report results with more than 20 second latencies (e.g., map-reduce jobs). I target online, data-intensive workloads in my research,

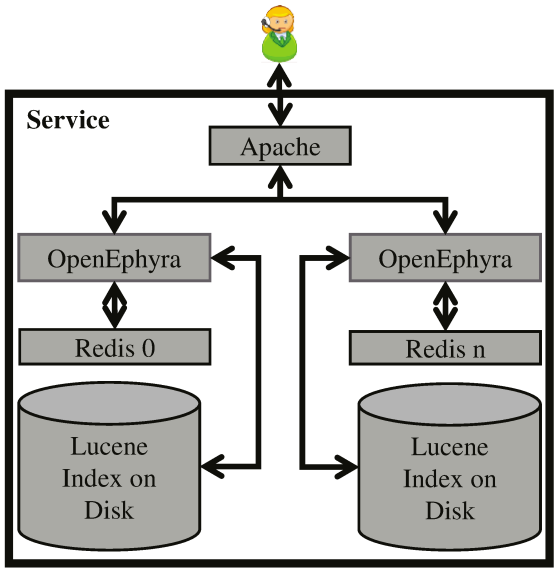

Figure 1.3: Example online, data-intensive service processes unstructured data from memory cache and disk to find answers to natural language questions.

covering both services with short expected latencies and longer-running workloads. Regardless of expected service level objective (SLO), these workloads process large amounts of data in the cloud. The online, data-intensive services I have used in my work are available from [73]. The workloads I have used in my work which expect longer service times come from the NAS Parallel Benchmarks [53].

Overhead from online collection of workload characterization data has to be balanced with its value. Value is a broadly defined term referring to the business benefit of data. Of all data that was generated in 2013, perhaps 22% may have been usefully analyzed, had it been

tagged [158]. However, less than 5% of all data generated in 2013 was considered target-rich [158]. More concretely, data has value in business when it directly increases revenue or decreases costs. Data about system operations can have reduce costs by improving performance and lowering power expenses. Engineers at Facebook use Scuba to collect, store, and analyze trace data regarding the quality of service reported by users [1]. Problems in Facebook software and infrastructure are identified using Scuba before more users are impacted, decreasing the cost per infraction.

Thesis Statement:
Systems support, in the form of OS tools and middleware, allows a wide range of workloads to respond to resource needs by adaptively tuning their quality of results and workload characterizations while preserving high throughput.

I trade reduced accuracy for lower overhead from workload characterization. Trading 2% accuracy can reduce the overhead of workload characterization by 20%. Reducing the expected accuracy allows characterization of workloads that would otherwise be too costly to profile to completion. My work is split into three sections. The first uses offline profiling to determine task sizes for data-intensive, subsampling workloads and reduce profiling time for finding peak power at different numbers of active cores. The second introduces limited caching to improve latency and allow answer quality to be obtained online. The last piece of this work introduces a design for studying workload characterization in colocation environments.

In the first section of my work, I use offline profiling to analyze the overhead attached to job creation and task sizing within the Map Reduce framework. The key observation, presented in Chapter 2, is that when a randomly accessed task falls out of cache, this has a larger impact on total job latency than the overhead involved in creating more, smaller

tasks. In Chapter 3, I present a quantitative study of power, most importantly that power phases remain similar regardless of the number of cores used. Based on this observation, I explain my algorithm for adaptive peak power profiling, which suggests profiling on the maximum number of cores to determine how long to profile on other numbers of active cores. With just a slight relaxation of accuracy, the amount of time needed to profile is greatly reduced.

In the second section of my work, I explore caching as a way to reduce overhead. In chapter 4, I explore what can be done for a single service using the extra provisioning capacity (5%) reserved for the operating system. The use of replication for predictability on just the portion of cached data accessed last can reduce latencies by up to 12%. I take this further in chapters 5 and 6. Chapter 5 introduces the answer quality metric for use with interactive natural language processing workloads in an offline setting. Chapter 6 introduces Ubora, which allows resource managers to obtain and manage answer quality online for interactive, data-intensive workloads. Rather than characterizing every request for answer quality, resource managers set the rate at which queries are sampled, trading accuracy for reduced overhead.

Finally, chapter 7 in the third section of my work examines workload characterization in a colocation environment. My design suggests speculatively deploying a running workload to a colocation environment, and then combines the techniques used for answer quality and power study in order to characterize a workload online. I implemented my design as Quikolo, a platform that speculatively deploy workloads in Docker images to a Kubernetes environment, to study the overhead impact of number of features and duration on characterization accuracy. My Quikolo platform outputs a recommendation to the user regarding

whether that workload will increase latency or get fewer SLA violations if it migrates to the speculative location.

Chapter 2: Managing Tiny Tasks for Data-Parallel, Subsampling Workloads

Internet services and mobile devices have generated large amounts of data. Indeed, 90% of all data has been produced in the last two years [136]. Data will continue to grow as other types of data collection become popular. For example, genome sequencing has become 10,000X cheaper over the last 8 years [109]. The genetic sequencing of all American adults could produce an additional exabyte of data. Big data is becoming too large to process exhaustively, especially for interactive workloads that must produce results quickly. 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 causing poor locality can significantly degrade 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 chapter, we set up a data-parallel platform that supports tiny tasks and speeds up subsampling. We used a two-step approach. First, we profiled existing platforms for tiny-task scheduling overhead. We compared three Hadoop configurations and BashReduce (a lightweight implementation of the map-reduce paradigm [39]). 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 [145]. 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 [175, 144, 115].

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 [120, 129]. 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 [168, 2, 174].

Our Contributions: This chapter focuses on interactive, data-parallel workloads [71, 144, 114, 115, 120, 96]. Map and reduce tasks within these workloads complete quickly, relying on efficient processing and on low scheduling overhead [115]. 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 chapter, Section 2.1 describes subsampling workloads and their locality issues. Section 2.2 explains our benchmarking study on scheduling overheads and presents our modified BashReduce scheduler that supports task sizing. Section 2.3 presents experiments. Section 2.5 discusses related work.

2.1 Subsampling Workloads

Figure 2.1 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 2.1.

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

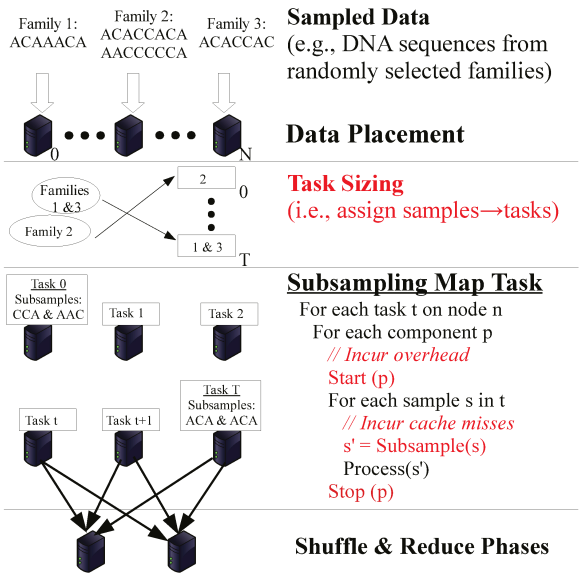

Figure 2.1: Data flow for a data-parallel subsampling workload.

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, 168]. Our research is orthogonal to this research.

A task comprises the software components used to process samples (p in Figure 2.1). 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.

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 chapter 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 [175], RDD [172], Data Cube [108], Sparrow [115], and [71]. These workloads may support interactive analysis of scientific data, personalized advertising, sentiment analysis, or real-time trace studies [175]. 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 [174].

2.1.1 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

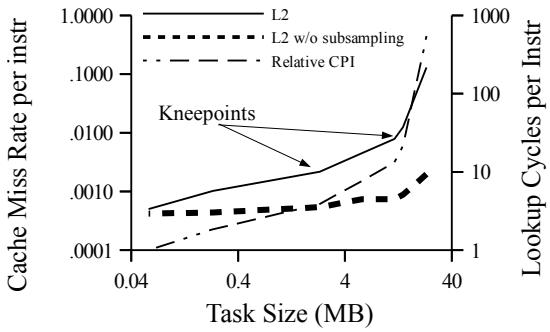

Figure 2.2: L2 misses per instruction and cycles per instruction across task sizes in EAGLET.

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 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 observed cache misses.

Figure 2.2 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 [145]. 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 2.2, 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 [88] 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 [28]. 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 2.2: *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 [118]. The secondary axis on Figure 2.2 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.

2.2 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 [120]. 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.

2.2.1 Job- vs Task-level Recovery

Hadoop was designed to process multiple petabytes spread over $10^4 - 10^5$ nodes [164], 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 [172, 115, 71] and 2) their goal is often to compute results from iterative or incremental changes [96, 99].

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

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 [39]	Unix Utilities	No	No	No

Table 2.1: Platforms benchmarked for this paper

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 [175]—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 [115, 71, 174], we set $N = 100$. We set $mttf = 4.3$ months from [120, 36]. 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.

2.2.2 Platform Selection

We measured scheduling overhead for the platforms shown in Table 2.1. Here, we describe the salient features of each platform. More details are can be found in Section 2.3.

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 [39]. 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 BashReduce. BashReduce also elides a global distributed file system (HDFS). Managers partition data and tasks access only the local file system.

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 2.1. 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

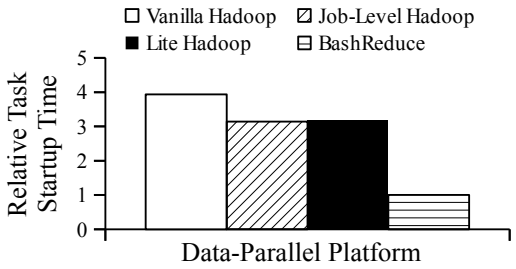

Figure 2.3: Relative time to start 1 task on each core by platform

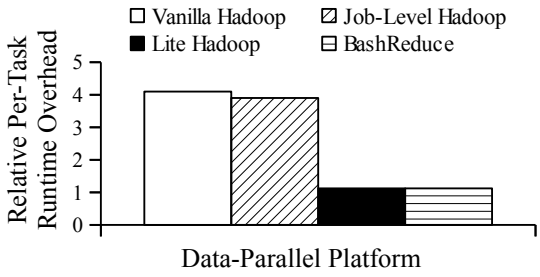

Figure 2.4: Runtime overhead of each platform relative to native Linux

running time on Hadoop). Figure 2.3 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 2.4 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.

2.2.3 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. 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 2.2).* 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 2.5 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 2.2, we plot the aggregate input data size against cache misses per instruction. We modified our BashReduce platform to group samples into tasks

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();
}
```

Figure 2.5: Java code for offline kneepoint detection and task sizing implemented within BashReduce.

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

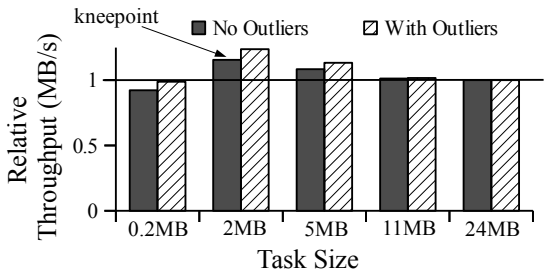

Figure 2.6: Impact of our kneepoint algorithm on runtime

have large outliers [7, 71]. 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. A further avenue to follow in the future would be to develop 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 2.6 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 2.6). 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 [7, 144]. 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 are not known. Our experiments in the next section show that kneepoint selection is insensitive to small errors.

2.3 Experimental Setup

We set up two subsampling workloads. EAGLET [145], 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 [9]. 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 [110, 174]. 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 [174] 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 [39] 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 2.5. **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.

	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 2.2: Hardware used in our studies.

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 2.2. 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.

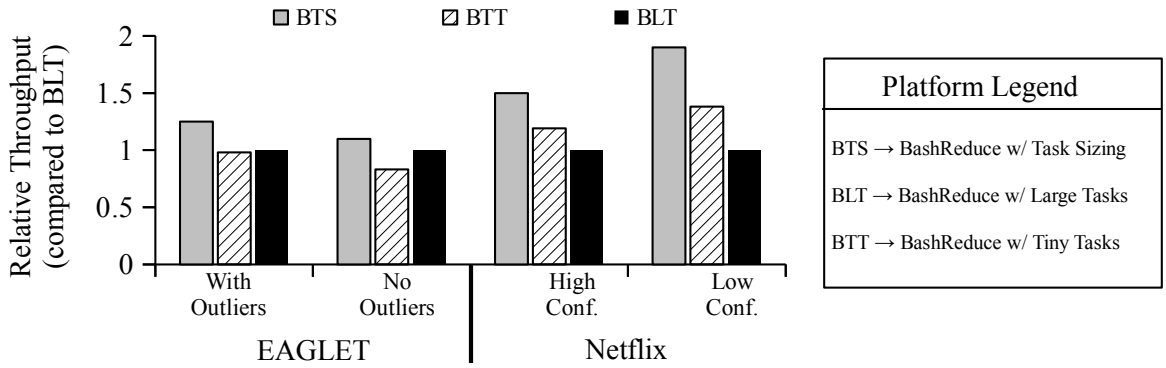

Figure 2.7: BTS speeds up both EAGLET and Netflix workloads relative to BLT and BTT. Tests ran on hardware type 1. The rightmost table provides acronyms for all of the platforms referenced in this section.

2.4 Experimental Results

Figure 2.7 compares the BashReduce setups. For this test, we used 6 nodes of hardware type 1 (See Table 2.2). 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 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 2.8 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

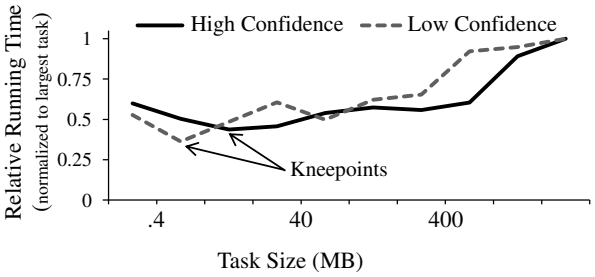

Figure 2.8: Kneepoints in the Netflix subsampling workloads on BashReduce.

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 2.7 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 [164]. We compared the throughput of BTS to three Hadoop setups across different job sizes. For these tests, we ran the EAGLET

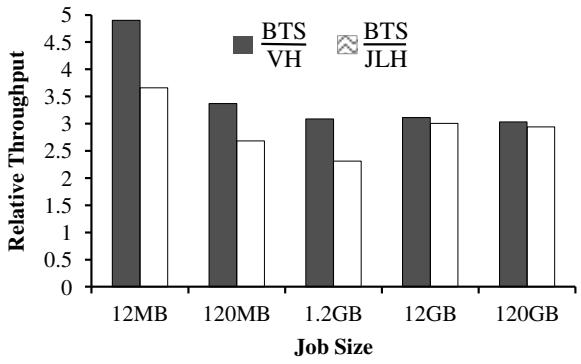

Figure 2.9: Comparison of BTS to VH and JLH.

subsampling workload on type 2 hardware, varying job size. We changed the job size by adding synthetic families to the Bi-Polar data.

Figure 2.9 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 [88] to capture L2 and L3 cache misses, instruction counts, accesses

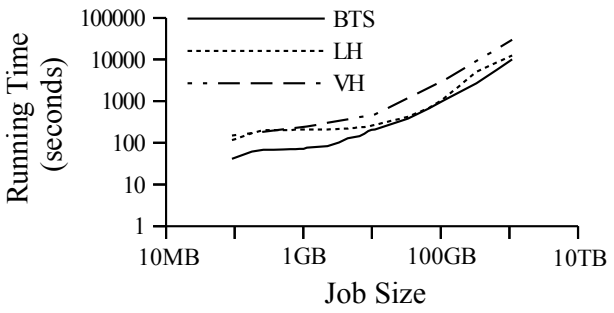

Figure 2.10: Comparison of BTS to VH and LH in terms of running time. Note log-log scale.

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 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 2.10 shows BTS's speedup over VH. These tests used 72 cores

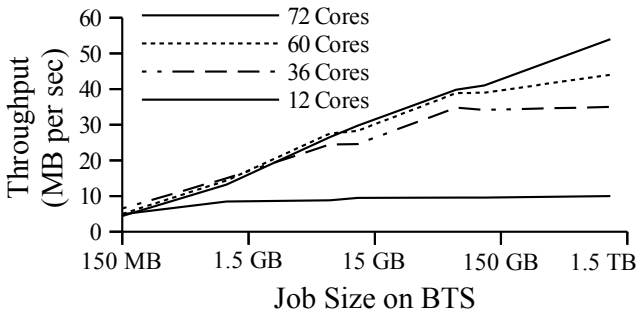

Figure 2.11: EAGLET on BTS as number of cores changed.

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 [120] and 24 Mb/s [129]. 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 2.11 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.

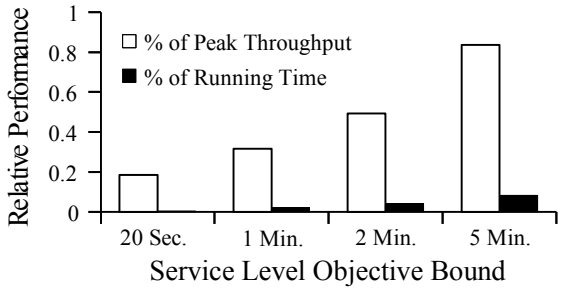

Figure 2.12: The throughput and running time of EAGLET on BTS clusters scaled to efficiently meet service level objectives.

The 72-core test (i.e., 6 type 2 nodes) produced results at 45% of network capacity. In Figure 2.11, 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 [174, 175, 16, 144]. For data processing workloads, a job's running time depends on its 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 2.12 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

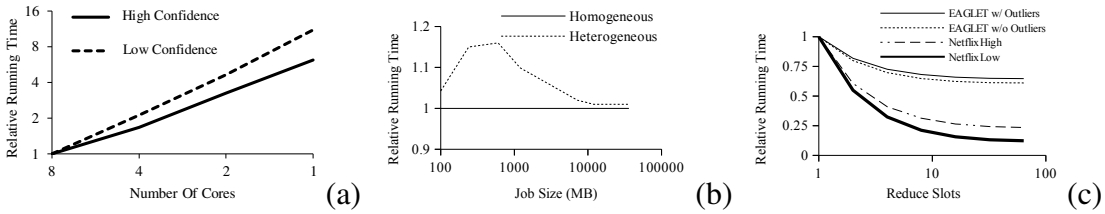

Figure 2.13: Running time on BTS (a) Netflix workload as cores scale on Type 3. (b) Netflix workload as job size increases. (c) Network resource demand increases.

bounds. Figure 2.12 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 2.2). 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 2.13(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 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 [174] 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 2.13(C) highlights the results. With EAGLET, secondary genetic analysis is compute intensive [120]. As a result, adding reduce tasks quickly exhibits diminishing returns. The Netflix workload, however, can speed up at the reduce stage.

2.5 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 chapter 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 [115, 114] presents a data-parallel version of power-of-two load balancing [102] that allows independent schedulers to avoid transient delays. Each node's operating system and background jobs also cause transient delays. Replication for predictability [71, 144, 7, 23] 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 [21]. Mizan [77] 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. [175] 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 [141]. GreenHadoop and GreenSlot [47, 48] also create accurate performance models. However, their focus was exploiting intermittent renewable energy [142]. AMAT (average memory access time) is a simple model that makes a strong point: faster storage can significantly decrease execution times. RDD [172], Data Cube [108], Pig [175], and [71] 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. [157] provide a framework to compare caching and partitioned hybrid architectures. hStorageDB is one such hybrid system [92].

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. [96] propose language support for differential dataflow, a paradigm that allows programmers to specify incremental structure in their programs. RDD [172] users can call functions on cache misses, allowing for certain types of incremental workloads. Waterland et al. [163] 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 [15, 20, 70].

Chapter 3: Adaptive Power Profiling for Many-Core HPC Architectures

Many-core architectures are now pervasive in HPC. In 2009, 51.2% of HPC workloads deployed on Jaguar at Oak Ridge National Laboratory used more than 7832 4-core nodes, and Jaguar has since expanded into Titan, with 18688 16-core compute nodes and associated GPUs [154, 155]. Soon, 72-core nodes will be available for HPC environments [137]. Unfortunately, few workloads use every core effectively. For example, PARSEC benchmarks achieve 90% of the speedup provided by 442 cores using just 35 cores [30]. Our tests with NAS workloads on Intel Xeon Phi confirm these results, using 32 cores provides 85% geometric-mean speedup relative to 61 cores.

Workloads that execute threads on every core have large and inflexible power needs. Further, their peak power needs can be significantly larger than average power needs. To be clear, peak power is the largest aggregate power draw sustained for at least a tenth of a second [40]. Large peak-power needs present a challenge for modern HPC centers that operate under a myriad of increasingly tight power caps. Circuit breakers enforce hard power caps that safeguard electrical equipment [40]. Emerging demand-response systems enforce soft power caps that shape power usage over time. For example, workloads that

cap their peak power consumption during mid-afternoon hours provide significant cost savings [165]. Running Average Power Limit (RAPL) enforces soft power caps on low priority workloads [128, 57, 173].

HPC workloads can lower their peak power needs by restricting software threads to a subset of available cores. This approach is called *core scaling*. Core scaling reduces the dynamic power used to access memory, manage on-chip caches, and operate processors. With core scaling, cores that do not execute software threads have lower dynamic power needs and transition into low power operating modes. However, core scaling can increase peak power for the cores used to execute threads, because each active core executes more instructions. Resource contention can cause data movement between threads that increases power usage. Higher power usage can lead to increased temperature and potentially higher system failure rate [149, 112, 52, 13, 155].

HPC schedulers use workload profiles to allocate resources to jobs at runtime, and can change provisioning as the workload progresses [111, 24]. Workloads decide which active cores will execute threads at runtime. However, resource contention between threads is dynamic. Consequently, a wide range of runtime factors affect a workload's power usage. Offline models of power usage miss these factors [44, 14]. These models can capture the *potential* effects of core scaling but may not accurately reflect the *actual* effects.

For this chapter, we studied the effects of core scaling on peak power. Our experimental design measures power usage during workload execution. As such, our measurements capture power usage caused by dynamic data movement. Specifically, we applied core scaling to HPC workloads running on modern architectures and widely used parallelization

platforms. We used RAPL to trace instantaneous power usage during execution. Key findings from our study are listed below:

1. Power savings from core scaling varied across workloads and architectures.
2. Workloads that exhibited similar peak-power savings on one architecture were often unalike on another architecture.
3. Workload phases trigger power spikes at similar execution points across core scaling settings. Often, these power spikes occur early in execution. After 40% of a workload's execution, peak power up to that point predicted final peak power within 5% for 13.8% of the configurations studied.

We focus our study on peak power profiling, which partially executes a workload on a target architecture. Peak power during partial execution is used to model final peak power. Accurate peak-power models are critical for schedulers and capacity planning [128]. Our study showed that peak power changes significantly under core scaling. Further, the changes are not easily modeled analytically. Thus, we explored the challenge of peak-power profiling across core scaling settings.

State of the art profiling approaches partially execute workloads for a fixed sampling duration (e.g., 5 min) [24]. We developed adaptive peak-power profiling. Our approach accepts two inputs: sampling duration and desired accuracy. First, it partially executes the workload for the sampling duration and collects a power trace. To estimate peak power under other settings, it executes the workload until either 1) the workload causes a power spike or 2) the sampling duration is reached. Our approach reduced profiling time by more than 40% for most applications while allowing for scaled estimation accuracy.

The remainder of this chapter is as follows. Section 3.1 describes the methodology for our study. Section 3.2 characterizes peak power as workloads and architectures changes. Section 3.4 characterizes peak power relative to instantaneous power early in a workload. Section 3.5 presents our adaptive profiling approach driven by our results. Section 3.6 presents related work.

3.1 Experimental Methodology

Core scaling can increase or decrease peak power. As core scaling idles more cores, more cores have reduced their power consumption compared to the active cores. On the other hand, reducing the number of active cores causes the active cores to sustain larger work load which can increased power needs. Our experimental design characterizes the relative importance of these diametric forces. In this section, we first provide details on our power measurements. Then, we discuss independent secondary factors that affect core scaling. These are listed in Table 3.1.

3.1.1 Power measurement

In our experiments, we used RAPL to measure power usage during workload execution. RAPL stores power measurements per CPU socket in Linux Machine State Registers (MSR). The open-source libRAPL library provides an API to access MSR [78]. Specifically, we measured on-core, and uncore (memory), and total energy per CPU socket, since initialization. We converted total energy into power by associating each libRAPL call with its corresponding wall-clock time. We issued libRAPL calls every 100 milliseconds. For Xeon Phi, we read power from micsmc [79] instead of RAPL.

Factor: Architecture

Values	Features
i7	4 cores, 256 KB L2, 8 MB L3, 3.4 GHz, 95 W TDP, 32 nm
Sandy Bridge	8 cores, 256 KB L2, 20 MB L3, 2.6 GHz, 115 W TDP, 32 nm
Xeon Phi	61 cores, 512 KB L2, N/A, 1.05 GHz, 225 W TDP, 22 nm

Factor: Parallelization Platform

Values	Features
MPI	Message passing interface, limited support for shared memory
OMP	Open Multi-Processing platform provides extensive shared memory support

Factor: Workload

Values	Features
CoMD	N-body molecular dynamics. Frequent inter-thread communication
Snap	PARTISN workload used at LANL. Iterative inter-thread communication
MiniFE	Finite-element workload composed from 4 parallelizable kernels
NAS benchmarks	Widely used benchmark suite (CG, EP, FT, IS, LU, MG)

Table 3.1: Secondary factors in our experimental design. Thermal design power (TDP) is maximum power consumption.

We verified our approaches with Like I Knew What I Was Doing (likwid) [156], an open-source tool for reading hardware counters. Likwid measured total energy for a whole workload execution. However, likwid verified the energy measurements from our approach within 1% on average.

3.1.2 Architectures

We experimented with 3 architectures: an i7-2600K, Sandy Bridge (Xeon e5-2670) and Xeon Phi (5110P). The specifications for each architecture are shown in Table 3.1. Each architecture used low-power operating modes when cores idled. For the i7 processor, we set up the On-Demand CPU Governor, a kernel driver that adjusts operating frequency. The maximum frequency was 3.4 GHz. The minimum frequency was 1.6 GHz. For the Sandy Bridge processor, we enabled processor controlled P-states. The maximum and minimum operating frequencies were 1.2 GHz and 2.6 GHz, respectively. The Xeon Phi used the micsmc controller with P-states and C3 package [79]. The C3 package can reduce power usage 45 W when the whole processor is under used. P-states reduce power based on each core's usage.

3.1.3 Platforms

Message Passing Interface (MPI) and OpenMP (OMP) platforms provided mechanisms for core scaling. Specifically, we specified the number of active cores when we launched workloads. Both platforms exhibited similar peak power when we ran the micro-benchmarks. The average difference was less than 2%. However, the platforms achieve parallelism using

qualitatively different approaches. MPI uses message passing with limited transparent support for shared memory. OMP uses shared memory. We hypothesized that these platforms could cause different peak power under core scaling.

We used likwid to read hardware counters related to data movement. Specifically, in this chapter, data movement refers to data path transfers between: L1 and registers, L1 and L2, L2 between cores, L2 and L3, LLC and memory. Data path transfers include loading data into the cache, writing data back to memory, and coherence related transfers. We use data movement throughout the chapter to explain the root causes for the effects of core scaling.

3.1.4 Workloads

For our research, we used the set of NAS Parallel Benchmarks available from the NASA website [53]. To ensure fairness, we only used benchmarks which could run on a number of cores equal to a power of 2. These six benchmarks are Conjugate Gradient (CG), Embarrassingly Parallel (EP), discrete 3D fast Fourier-Transform (FT), Integer Sort (IS), Lower-Upper Gauss-Seidel solver (LU), and Multi-Grid on a sequence of meshes (MG). These benchmarks, commonly used in the HPC community, run on a number of cores that is a power of 2. We used application size B unless otherwise specified. Application size B is the appropriate size to run on a single node. Integer Sort was written in C, while the other 5 NAS benchmarks were written in Fortran.

The Conjugate Gradient algorithm (CG) is known for its irregular memory access patterns [53]. A member of the unstructured grid type of computations, it approximates the smallest eigenvalue of a positive definite symmetric matrix. The matrix in this data set is

large and sparsely populated. The Embarrassingly Parallel algorithm (EP) tests the limits of floating point performance, and has no significant communication between cores. The Fourier-Transform algorithm (FT) solves a 3D partial differential equation, and is representative of spectral computations. Its primary feature is all-to-all communication. The Integer Sort algorithm (IS) is known for random memory access. This sorting algorithm tests the speed of integer computation. The Lower-Upper Gauss-Seidel solver (LU) uses a system of nonlinear partial differential equations to simulate computational fluid dynamics. LU contains a limited degree of parallelism compared to these other benchmarks. The Multi-Grid algorithm (MG) is memory intensive and features long and short distance communication [11].

In addition to the NAS Parallel Benchmarks, we also use three kernels used to simulate real high performance computing workloads running on Titan, named CoMD, miniFE, and snap. CoMD is a classical molecular dynamics application. MiniFE is a finite-element mini-application which assembles a sparse linear system using a conduction equation, then solves this sparse linear system using a conjugate-gradient algorithm. The final result is then compared to the analytic solution [84]. Snap is a proxy application for modeling modern discrete ordinates neural particle transport application performance. Snap mimics the communication patterns, memory requirements, and computational workload of PARTISN, which solves the linear Boltzmann transport equation.

3.2 Observations on Power Consumption

In this section, we present observations from our study of the effect of increasing the number of cores on the power consumption on different architectures and workloads.

Observation 1. *Different architectures observe significantly different relative increase in power as the number of active cores increases. Reasons for this include the power consumption of other on-chip resources and built-in support for dynamically managing the power consumption of those resources based on the activity.*

We investigated how only increasing the core counts affects the power consumption by running a synthetic benchmark. This synthetic benchmark stresses the processor registers only and does not access any level of cache. The floor measurements in Figure 3.1 show the relative increase in power found with this simple synthetic benchmark. The relative increase in power is highly dependent even for such a simple synthetic benchmark. For example, Intel i7 and Sandy Bridge platforms observe a 1.9x and 2.15x increase respectively in the power at their maximum number of core compared to a single core count case, while such an increase in Intel Xeon Phi is only 1.37x. This indicates that as the number of cores scale for a processor core intensive workload, other resources (e.g., caches, interconnect) may also observe increased power usage in certain architectures. One of the reasons for this is that bandwidth to DRAM and L3 cache are dependent on frequency in the Intel Sandy Bridge architecture [131].

L1, L2, and L3 caches also affect power usage. We created micro-benchmarks that targeted each layer. Specifically, each micro-benchmark repeatedly accesses data stored in the targeted cache layer but not in the layers above it. Our micro-benchmarks capture the power caused by fetching data from the targeted layer. For example, a fetch from L3 causes data movement in the L1 and L2 layers. A fourth micro-benchmark repeatedly accessed data from memory.

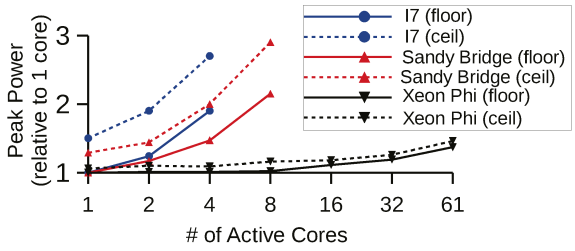

Figure 3.1: Peak power of workloads that repeatedly access L1, L2 and L3 caches and memory. Y-axis is relative to power consumed on 1 core by the workload with minimum (floor) peak power.

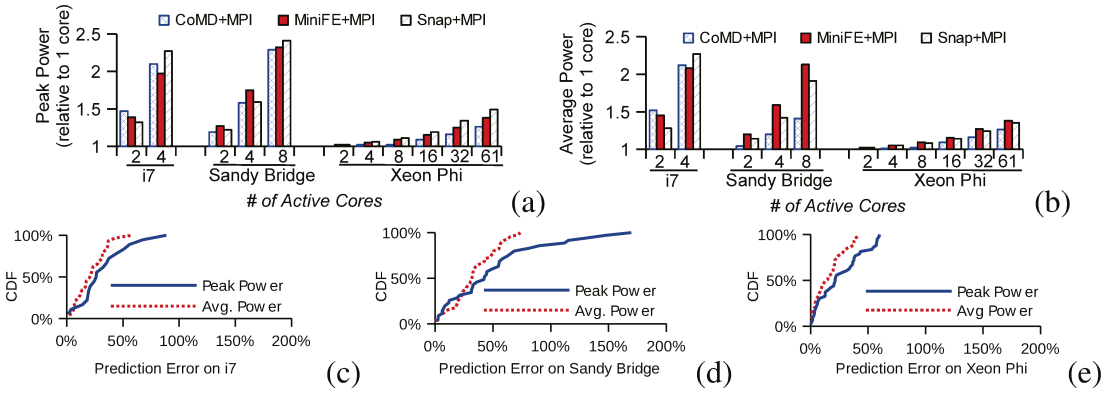

Figure 3.2: Power and prediction error ranges vary across architectures.

The ceiling measurements in Figure 3.1 shows the effect of core scaling on the power consumption of our cache and memory micro-benchmarks. One may expect that the memory micro-benchmark will have provided a ceiling for each of our experiments, but this is not always the case. The micro-benchmark that targeted memory provided a ceiling only at

lower core counts. On the Sandy Bridge when all cores were active, the microbenchmark that accessed L3 cache consumed 16% more power than the one that targeted memory. This reflects bandwidth saturation in the microbenchmark that accessed memory; Sandy Bridge automatically lowered operating frequencies when cores idled waiting for memory. The i7 architecture also showed evidence of bandwidth saturation as the number of active cores increased. On the i7 architecture, the memory-targeting microbenchmark only consumed the most energy at a single core. The L3-targeting microbenchmark consumed 1% more power at 2 cores than the memory-targeting microbenchmark. The L2-targeting microbenchmark consumed 14% more power at 4 cores than the memory-targeting microbenchmark and 7.6% more power than the L3-targeting microbenchmark. Even on the Xeon Phi architecture, the 8 core experiment showed that the LLC-targeting microbenchmark used 2.7% more power than the microbenchmark that targeted memory. At larger core counts, the LLC and memory microbenchmarks peaked at the same power consumption on the Xeon Phi.

Core scaling has the greatest effect on peak power on i7 and Sandy Bridge, since the ceiling power at maximum active cores was as high as 2.7X and 2.9X the floor power on one core. In contrast, on the Xeon Phi, using 61 cores cost at most 1.45X the floor power of a single core. The Xeon Phi uses a ring-based inter-connect that is fully powered if just 1 core is active. This reduced the power savings provided by core scaling.

Core scaling caused larger increases in peak power as the number of active cores increased. This was especially true for the ceiling micro-benchmarks that caused large amounts of data movement on each access. Note, the micro-benchmarks do not reflect upper or lower bounds on peak power. Workloads can stress multiple components at the same time.

Observation 2. *There exists significant variation in relative increase in power consumption across workloads on different architectures. In fact, the same workload may exhibit significant variation in peak power consumption between platforms with increasing core counts.*

We found this observation while studying how power consumption varies for different numbers of active cores across different real MPI workloads. Figure 3.2(A-E) present the peak and average power consumed by different MPI workloads at different core counts on different architectures. These results show a couple of interesting trends. We observe that there exists a significant amount of variation in power consumption across workloads on all architectures. As the core count increases, the relative difference in the power consumption across workloads varies significantly as well. For example, on eight cores of the Intel Sandybridge platform, there is more than 20% variation in peak power consumption across all tested workloads, while such a variation is very limited at single core count (less than 5%). This variation is even more pronounced for average power consumption. These results indicate that workloads may benefit significantly by reducing the numbers of cores used, both in terms of peak power and average power consumption. However, such a variation is highly architecture dependent. For example, we observed less than 10% variation in peak power consumption across workloads on Intel Xeon Phi platform even at very high core count. This occurs in part because there is only one clock source in the coprocessor, so Intel Xeon Phi runs all active cores at the same frequency. Regardless of how much bandwidth the application requires, the Intel Xeon Phi always enables full bandwidth if even a single core is active [62]. Therefore, the only separation that occurs in the peak power between different benchmarks is in the processor frequency requirements.

Finally, we also observed that relative increase in power consumption for the same workload changes significantly across architectures. As an example, MiniFE has the highest relative increase of peak power on the Intel Sandy Bridge architecture of the workloads shown, but has a much lower relative increase in peak power on the Intel Xeon Phi compared to Snap.

3.3 Predicting Peak Power using Reference Workloads

In Section 3.2, we observe that there may be some variance in peak power across workloads and architectures (Figure 3.2). Given the limited variance in some cases, it is intuitive to investigate if one workload can be used as a reference for predicting the peak power of other workloads. Clearly, this approach has its practical limitations. For example, it may not be possible to examine all the possible reference workloads and identify the most suitable reference workload in a production environment. Identifying suitable reference workloads will require knowing the intrinsic characteristics and properties of workloads that affect the power consumption. Unfortunately, knowing such intrinsic properties and quantifying them may require high overhead profiling. In some cases, this may not be possible at all because production workloads may be mission critical (or proprietary).

Nevertheless, we explore this approach to understand the limits of peak power prediction when using other workloads as a reference.

Observation 3. *We observed that using one reference workload to predict peak power of other workloads can result in highly variable inaccuracy, especially across architectures. The best reference workload may even change across architectures.*

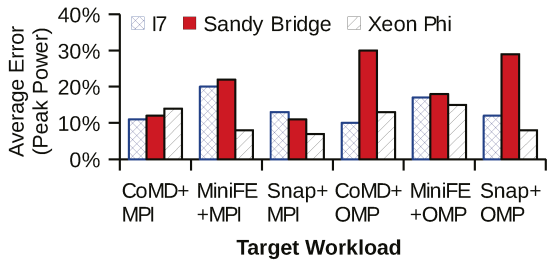

Figure 3.3: Average error in predicting peak power varies across architectures.

To discover this, we first investigated the effect of using a single workload as a reference for predicting peak power of multiple applications. While it is possible that each application may have its own best suited reference workload, we first tested the limits of this approach using a randomly selected workload as the reference. Figure 3.3 shows the error in predicting peak power using a randomly selected NAS benchmark as the reference to predict peak power of three applications on different architectures. We observe that the peak power error rates can be as high as 30% in some cases and can vary significantly across architectures. This indicates that the same reference workload can cause different amounts of error in peak power prediction on different architectures for the same target application.

However, as pointed out earlier, the best reference workload may be different for each application. Therefore, to further test the limits the peak power prediction from using reference workload, we found a pair of applications that have almost zero prediction error on one architecture (i.e., almost same peak power) and measured the difference in the peak power on a different platform. Figure 3.4 shows that while the NAS IS benchmark and

CoMD application have very similar peak power on Intel i7, their peak power on the Intel Xeon Phi differ especially at higher core counts. Second, Figure 3.4 also shows that CoMD MPI and OpenMP versions have very similar peak power on Intel Xeon Phi even at very high core counts, but significantly different peak power on Intel Sandy Bridge. These results illustrate that using the same reference workload may result in poor peak power prediction.

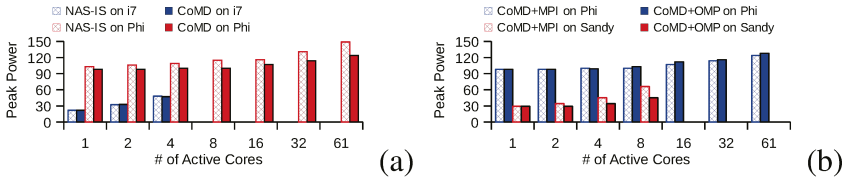

Figure 3.4: Peak power can be predicted using other workloads, with varying results. (A) CDF of error seen from all pairwise combinations of benchmarks. (B) Pairs with error under 15% on Intel i7 show that correlations do not continue across architectures.

3.4 Analyzing the Power Consumption Profile of Scientific Applications

In this section, we analyze the power consumption profiles of scientific applications on different architectures to derive insights about dynamic behavior of power consumption (Figures 3.6 - 3.8). We make several interesting observations about the dynamic power-profiles of these applications.

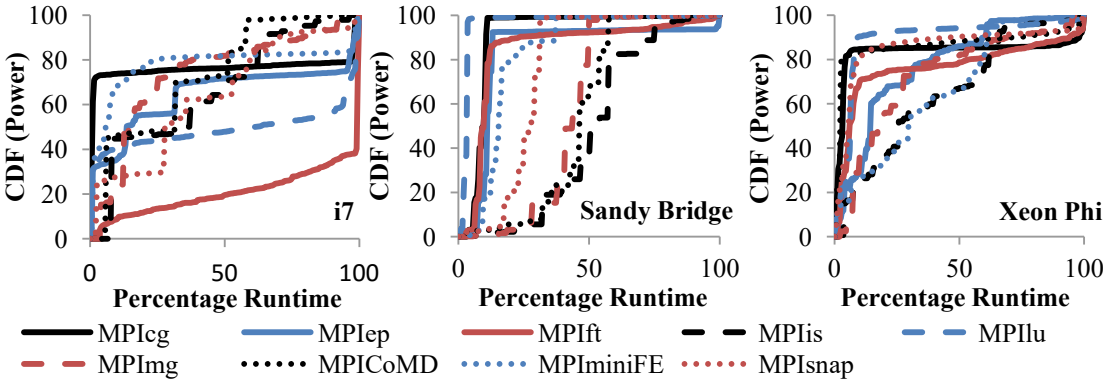

Figure 3.5: CDF of power consumption for different applications and architectures.

First, we investigate what fraction of the total execution time is spent at different power consumption levels. Figure 3.5 shows the cumulative distribution function of power consumption over the whole execution of applications on different architectures. We observe that different applications spend different amounts of time in or near their peak power consumption level. This characteristic is also highly dependent on the architecture even for the same application. For example, on the Intel Sandy Bridge architecture, MPI versions of CoMD, snap, and miniFE spend more than 50% of their execution time at a power consumption level higher than the 80% of the peak power of the given application. In contrast, these same applications spend only 20% of their execution time at a power consumption level higher than 80% of peak on the Intel Xeon Phi. Autonomic programs and schedulers often need to know how much of the time will be spent at k% or higher of the peak power load for efficiently provisioning the power distribution and estimating the cost of power and cooling. This analysis suggests that estimating how much time will be spent in or near

peak power requirements is not only application-dependent, but also highly architecture dependent.

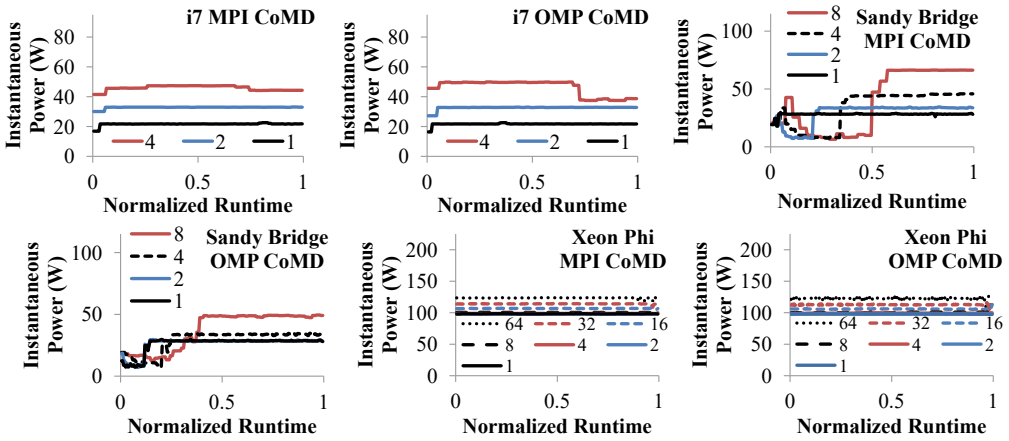

Figure 3.6: Power profile of CoMD application for MPI and OpenMP implementations on different architectures.

Observation 4. *We observed that the power consumption behavior changes significantly across application and can change significantly across architectures even for the same application. We also observed that there may be noticeable change in the power consumption characteristics across different implementations (MPI versus shared memory) of the same application. Both the underlying architecture and the application characteristics significantly affect how much time is spent in or near the peak power.*

As expected, scientific applications may exhibit distinct power consumption phases. However, we also observe that the power phases for a given application may change significantly as we change the underlying architecture. For example, MPI implementation of CoMD has

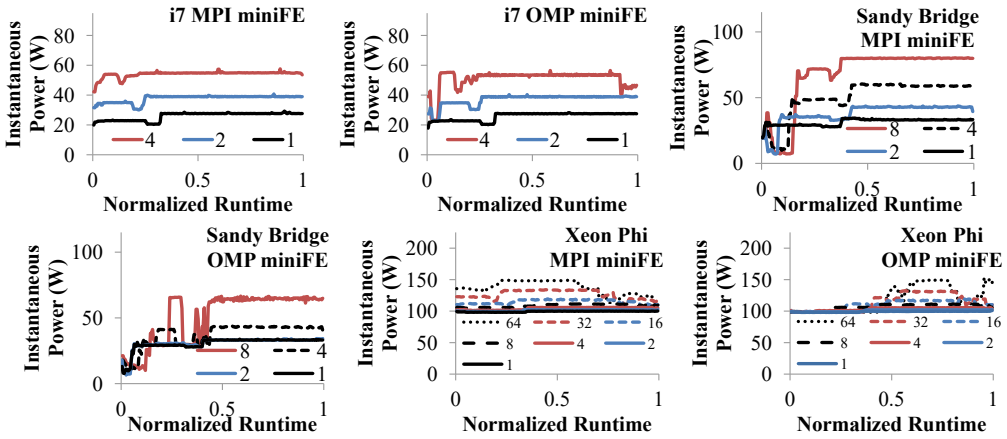

Figure 3.7: Power profile of miniFE application for MPI and OpenMP implementations on different architectures.

distinctly different power profiles on Intel i7 and Sandy Bridge platforms (Figure 3.6). Similar observations are true for MPI implementations of miniFE and snap applications. We note that this observation is not limited to the MPI implementation or multi-core platforms only. For example, OpenMP implementation of miniFE exhibits significant differences in the power profile on Intel i7, Sandybridge and Xeon Phi architectures (Figure 3.7).

Second, we observed that while MPI and OpenMP implementations of CoMD and miniFE applications exhibit similar power profiles on a given architecture, the third application, snap, shows significantly different power profiles between MPI and OpenMP implementations (Figure 3.8). This observation is pronounced on Intel i7 and Sandy Bridge platforms, albeit not so much on the Intel Xeon Phi platform. We observe that even the peak power differs significantly between MPI and OpenMP implementations. This indicates peak-power

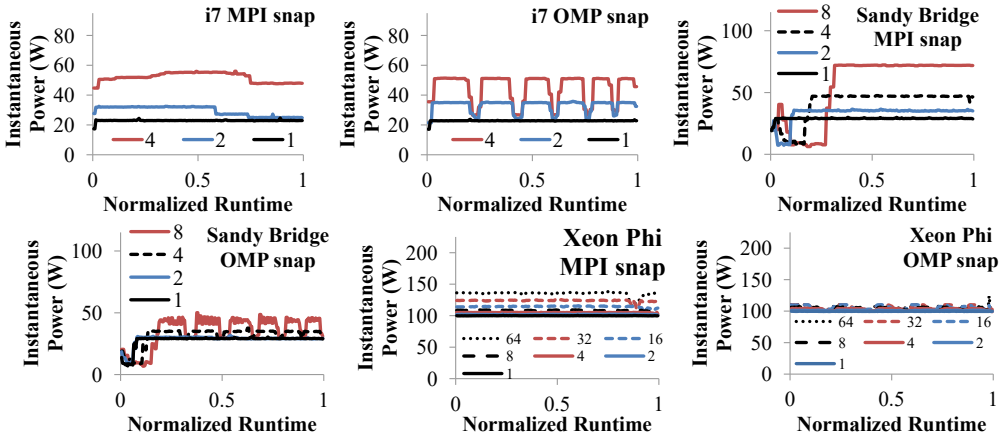

Figure 3.8: Power profile of snap application for MPI and OpenMP implementations on different architectures.

requirements are not only dependent on the architecture and underlying algorithm, but also on the platform (programming model) on which they are implemented (message passing versus OpenMP). For the applications we tested, we observed that MPI implementations usually resulted in higher peak power consumption (Figures 3.6 - 3.8).

Observation 5. *Interestingly, for a given application the power profiles are similar across different numbers of active cores on a fixed architecture given normalized execution time.*

This is an important observation that we exploit to estimate peak power across core counts for a given application and architecture pair in (Section 3.5). When we fix architecture, application, and platform, the power profiles with changing number of active cores display similar phases occurring at similar points in their normalized execution. A clear example of this is the OpenMP implementation of snap in Figure 3.8(D - F). Regardless of architecture,

clear phases can be seen, more distinctly at higher numbers of active cores yet still present at lower core counts. These phases are most distinct at higher core counts on the Intel Sandy Bridge and Intel i7 architectures because the variance in power with added cores is greater with these Turbo Boost capable architectures. However, these phases are still visible on the Intel Xeon Phi.

Power spikes from execution phases often occur early in execution, as can be explicitly seen in the power traces from Intel Sandy Bridge. We tested our workloads to see how far in their execution peak power occurs. In order to do this, we computed the maximum power so far seen in the workload at each timestep, and computed relative difference using this number and the peak power seen over the entire trace. This gave us a trace of peak power estimation error over the entire run of the workload.

We found that in many cases, profiling a workload on any core count for 40% of its execution resulted in peak power error below 5%. We show in Figure 3.5 the peak power error seen as the time spent profiling a workload increases for the MPI NAS benchmarks on Intel Sandy Bridge. Certain workloads can be profiled for a shorter time to achieve the same peak power error, and other workloads take longer on certain core counts. Overall, we found that peak power generally occurred later at higher core counts.

3.5 Adaptive Power Profiling

In this section, we present an algorithm to profile peak power across many core scaling settings. Consider the most widely used approach to profile a workload's power [24, 25]:

1. Choose how much of the workload to execute (i.e., k% of the running time).

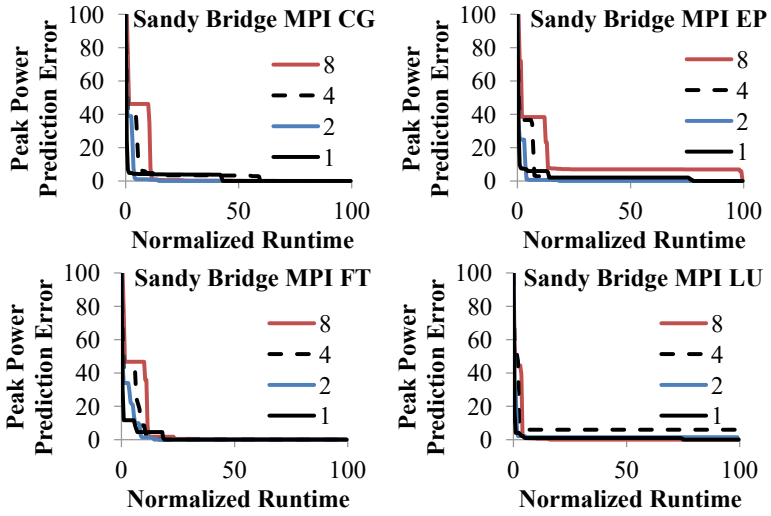

Figure 3.9: Peak power estimation error curves

2. Run the workload on the target architecture for k% of the running time.

3. During each run, collect power usage.

We call this approach *k% sampling*. To profile peak power under 5 core scaling settings, k% sampling must run 5k% of the workload. For $k = 5$, the delay to profile is 25% of total running time [24]. The previous sections showed that power phases occur at approximately the same point in a workload's normalized execution regardless of the number of cores used. Our approach uses power phases to reduce profiling time.

Figure 3.5 plots the absolute difference between peak power observed during execution and peak across the whole workload. The x-axis shows peak power observed at several execution points, ranging from 0.5% to 100% of the execution. Each line will eventually

converge to zero when the peak power is observed. The curves converge quickly if peak power occurs early in the execution, e.g., MPI FT. They also converge quickly if other high but not peak phases occur early in execution, e.g., MPI EP. Power phases ensure that the curves look similar across core counts.

Our approach profiles for k% at a single core count (by default the maximum core count). We then construct a single curve from Figure 3.5. We find the execution point on the x-axis where the error drops below a user provided threshold. This point is ($\hat{k}$%) and $\hat{k} < k$. *We then adapt our profiling to run workload for only $\hat{k}$% on the remaining core scaling settings.* The inputs to our approach are:

1. Sampling duration (k%)
2. Accuracy (i.e., maximum expected error)
3. Active cores for initial profiling run (by default, we assume the initial run will use all cores).

3.5.1 Our Profiling Method

Our method collects a power trace from a k% run of the workload at a single core count. This trace is represented by $PP(i)$ where $PP(i)$ is the power observed at timestep i. At each point in this trace, the peak power so far seen $PPmax(i)$ is calculated. We are guaranteed that the most accurate peak power over k% of the normalized execution of the workload will be found at $PPmax(k)$, where $PP(k)$ is the last reading in the trace. From this, we calculate a trace showing the expected error in estimating peak power $PPEC(i)$. For the

requested accuracy of $a\%$, our model uses $PPEC(i)$ to find the first point in the normalized runtime at this core count where an error less than $1 - a$ occurred.

$$PPmax(i) = max(PP(1), ..., PP(i)) \tag{3.1}$$

$$PPEC(i) = \frac{PPmax(k) - PPmax(i)}{PPmax(k)} \tag{3.2}$$

Data: PPEC[], k, a
Result: find sampling duration $\hat{k}\%$ to run at other cores
for $i \leftarrow 0$ **to** k **do**
  if $PPEC[i] < a$ **then**
    return i;
  end
end

Algorithm 1: This algorithm is used to find the duration to sample at other core counts.

We then collect a power trace from all other core counts for $\hat{k}\%$ normalized execution (Algorithm 1). Since we use the maximum number of cores by default, we conservatively assume that the profiled workloads are perfectly parallelizable. Running time doubles when core scaling halves the number of active cores. On every core scaling setting except the initial, our profiling runs for $\hat{k}\% \times \frac{\text{initialCoreCount}}{\text{currentCoreCount}}$. This time is likely longer than it would be if we knew the execution time on other core counts.

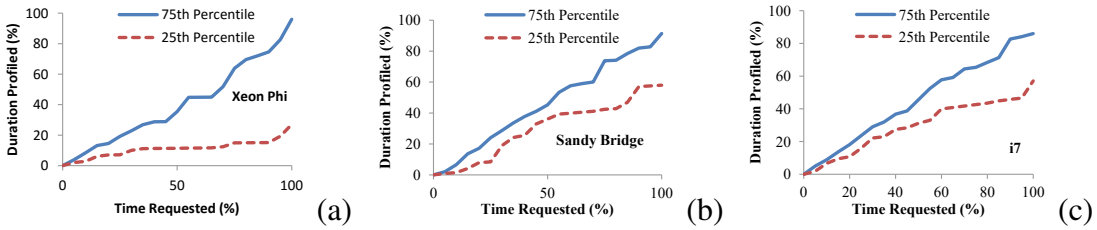

Figure 3.10: Duration spent profiling with our method as the requested time to profile increased. (a) Intel Xeon Phi (b) Intel Sandy Bridge (c) Intel i7.

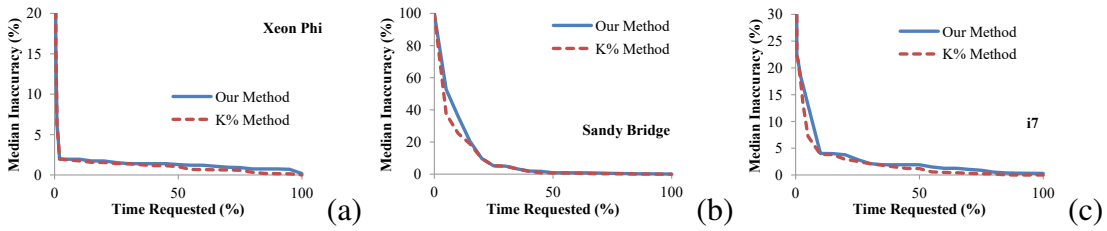

Figure 3.11: Median inaccuracy across all benchmarks for our method and k% profiling as the requested time to profile increased. (a) Intel Xeon Phi (b) Intel Sandy Bridge (c) Intel i7.

3.5.2 Evaluation

In our evaluation, we compare our method to k% profiling and prediction from similar workloads. These experiments used a fixed maximum core count, and assumed that a speedup profile was available so the correlation between k% and an actual running time was not a concern.

First, we compared the profiling duration at k% profiling to the profiling duration used with our max-core first method. For this experiment, we determined the percentage of workload

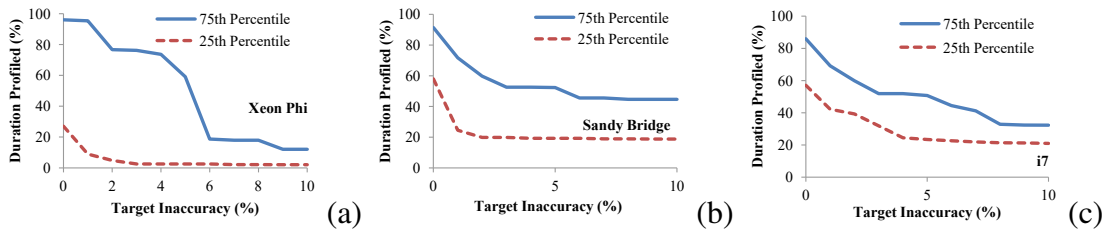

Figure 3.12: 75th and 25th percentiles of profiling duration across all benchmarks for our max-core first method as the approximation of peak power requested from the maximum core profile increases. (a) Intel Xeon Phi (b) Intel Sandy Bridge (c) Intel i7.

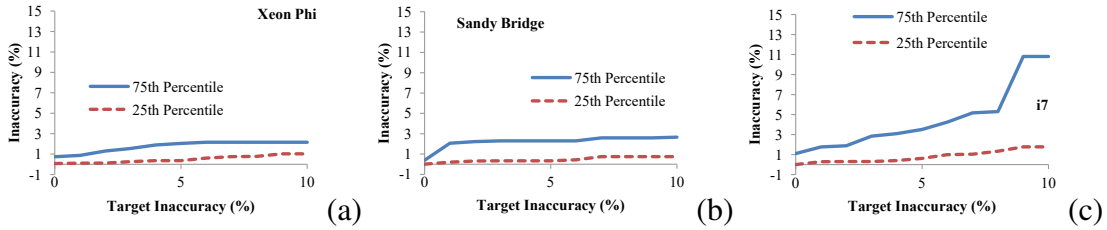

Figure 3.13: 75th and 25th percentiles of inaccuracy across all benchmarks for our max-core first method as the approximation of peak power requested from the maximum core profile increases. (a) Intel Xeon Phi (b) Intel Sandy Bridge (c) Intel i7.

to run by first profiling for k% on the maximum number of active cores, then finding the percent time at which the peak power was found. This percent time was then used as the profiling duration for the other core counts profiled. Our results in Figure 3.10 show the 25th and 75th percentile of total time used to profile all cores of a given workload. We found that we could reduce the amount of time profiling a workload across all core counts by up to 93% on the Intel Xeon Phi, 60% on the Intel Sandy Bridge, and 73% on the Intel

i7. The average time saved by using our max-core first method was 25% on the Xeon Phi, 12% on the Sandy Bridge, and 11% on the i7 architecture.

We also examined the effects of this selective reduction in profiling time on the profile's average inaccuracy (accuracy - 1). We found that on average, our max-core first method produced a profile within 0.3% of the profile produced with k% profiling for the Intel Xeon Phi. This same analysis found average difference of 1.5% on the Intel i7 and 3% on the Intel Sandy Bridge architecture.

We tested the behaviour of our max-core first model while changing the percentage of peak power we located in the profile of the highest core count. When we relaxed this requirement from 0% error in peak power estimation, we found that the resulting profiling time for other core counts still produced inaccuracy less than 3% for 75% of workloads, as shown in Figure 3.13. However, when this approximation percentage rose above 7% on the Intel i7, we saw the 75th percentile of workloads reach 5% or higher inaccuracy. This approximation dial allows us to trade accuracy for shorter profiling time. Figure 3.12 show that for our experiment with 100% profiling time requested, our max-core first model acquired these approximate profiles for most workloads using less than 60% of the profiling time requested on the Intel i7 and Sandy Bridge. The Intel Xeon Phi had a wider variation in estimated profiling time. The maximum normalized runtime used to get peak power stayed near 100% across architectures, but median normalized runtime dropped even with a 2% reduction in expected accuracy.

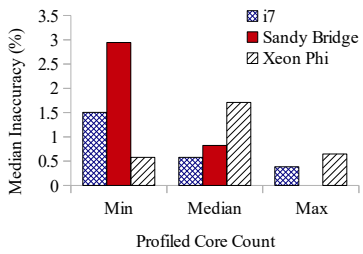

Figure 3.14: By architecture, inaccuracy averaged across core counts.

3.5.3 Corner Cases

We primarily tested our model using the maximum core count as the profiled core count, but it is possible to use a different core count for this. We experimented with profiling the minimum core count, the median core count, and the maximum core count to completion. We then profiled additional core counts using the percent time at which peak power was found on the fully profiled core count. The resulting peak power inaccuracy was averaged across all core counts for each workload. We show in Figure 3.5.3 the median inaccuracy across all workloads for each architecture. We found that the core count which offers the lowest median inaccuracy was the maximum core count, 4 for the Intel i7 and 8 for the Intel Sandy Bridge. The Intel Xeon Phi had a lower average inaccuracy on the minimum core count by 0.06%, but more than twice the number of workloads achieved 0% inaccuracy on the maximum core count compared to the minimum core count. Using the maximum core count, the greatest inaccuracy was 0.56%. Despite the intuition that using the maximum core count to determine how long to profile on other core counts, no core count was clearly preferable to the others in terms of running time.

3.6 Related Work

Researchers have studied the impact of core scaling on performance, and power consumption as the number of available cores on the multicore processors increases [147, 103, 44, 161]. For example, [147] proposed a feedback-driven approach to maximize performance by varying the number of threads at runtime based on data synchronization and off-chip bandwidth. Similarly, [103] propose approaches that can choose the optimal number of threads based on offline and online method. Core scaling is combined with and scaling of resources in [44] with a focus on power-constrained processors. [161] proposed a method to utilize concurrency levels and DVFS to achieve optimal energy efficiency configuration for a workload. The goal is to achieve maximum performance within a given power budget. In contrast to these works, our study is a unique empirical study of the effects of core scaling on peak power.

Peak power has also been explored by several studies, and it can be divided in two broad focuses, (1) Power capping (2) Peak power prediction/estimation. Works such as [80, 33, 31, 128, 57] propose models and dynamic techniques to keep power consumption under a budget. On the other hand, peak power is relatively less explored by the research community. Performance profiling and performance prediction for better scheduling decisions in datacenters has been explored by [24, 169, 72, 124]. A key challenge is profiling performance, power, and/or answer quality quickly in data centers that support heterogeneous hardware and software. Our study complements these works by proposing a peak power profiling and prediction framework. By combining the observations from peak-power profiling and core scaling, our proposed framework exploits the consistent nature of power usage across workload phases to provide accurate peak power prediction at a low overhead.

Chapter 4: Balanced and Predictable Networked Storage

Big data is often too complex for mere mortals. Graph processing [49, 82, 119], NLP [130], and data mining tools try to reduce big data to smaller but still useful nuggets. These workloads pull in large amounts of data, process it, and then return a smaller result. Pulling in the data is often the slowest part [120]. Loading 1GB from today's disks takes almost as long as it did 4 years ago. 10Gb Ethernet exceeds disk bandwidth by more than 10X, making it faster to access data stored in a remote node's main memory than to access it from local disk. As a result, network storage is used more often for big data workloads.

Big data workloads strive for balance, i.e., all nodes should be busy at all times. Well-balanced workloads achieve high throughput without wasting resources. For workloads that use networked storage, balance means there should always be a few backlogged accesses, but the backlog should not idle nodes in the data processing layer. One approach to achieving balance is to 1) measure typical storage access times, 2) measure the average access rate of each node that does data processing, and 3) size the data processing cluster according to the quotient of these numbers. In practice, this approach falls short, because access times in networked storage often have heavy tails. A few outlier accesses take much longer (100X) than typical accesses. These outliers cause delays in the data processing layer, delays that can not be recovered easily.

For this chapter, we studied slowdown caused by slow storage accesses in balanced map reduce systems. First, we compared access times from a real key-value store against exponential and Pareto Distributions. The Pareto was a better fit because of its heavy tail. Then, we modeled an access's *slack*, i.e., the smallest response time that would cause a delay in data processing. Finally, we used the Pareto to compute the expected delay caused by accesses that exceed their slack time.

Our model showed that outliers can slow down balanced map reduce by 70% when map tasks complete quickly (i.e., within 40ms). Slowdown decreases for workloads with longer map times and lighter tails. Storage capacity per map node also affects slowdown. Maps that need random access to big data spread across many nodes are vulnerable to slowdown. We concluded that these properties, short map times and random access to big data, often describe workloads that reduce big data, e.g., graph processing and stream sampling.

We extended our model to study replication for predictability, an old but seldom used approach to reduce the effects of outliers. We found that replication for predictability was most effective for short map jobs with large working sets, the conditions where outliers caused large slowdown. When maps complete quickly, replication for predictability prevented 12.5% of lost throughput while using only 5% of storage resources.

The remainder of this chapter is as follows: Section 4.1 discusses the trends and motivations in data processing that underlie this work. Section 4.2 presents our problem statement. Section 4.3 walks through our model that captures slowdown caused by outliers. Section 4.4 extends our model to consider replication for predictability and studies its cost effectiveness. Section 4.5 discusses related work.

4.1 Trends

Hadoop [151], Ceil [107], and Dryad [65] share a common trait: data pipelining. These data processing platforms try to keep disks, CPUs, and network links busy at all times. For example, Hadoop works best when data from a node's local disks is pulled in asynchronously while map and reduce tasks run concurrently. However, a node's local disks no longer offer the best performance [121, 120]. Today's disks support 800Mb/s whereas today's local area networks can support 10Gb/s. Networks will become even faster in the future as 40Gb/s Ethernet and hybrid electrical/optical switches are adopted.

When raw processing speed is the metric of merit, a 1 TB disk should be replaced with 16 64GB in-memory networked stores. On 10Gb Ethernet, the latter can achieve more than 300X speedup. Even on 1Gb Ethernet, a well-managed in-memory networked store can offer 20X speedup. The downsides for in-memory approaches are cost and power usage. Both increase quickly as data sets scale. When cost is also a concern, each node should support multiple disks with data striped across them. 16 disks accessed in parallel fall just below the throughput of 10Gb Ethernet [121]. However, 16 disks may not be enough in a few years. Further, the benefit of fast, random data access will make networked storage attractive.

Figure 4.1 depicts data flow and bandwidth when map reduce uses networked storage instead of node-local disks. In this chapter, we will assume networked storage is in the form of in-memory key-value stores, e.g., MemCached [29], but our ideas extend broadly to other types of stores. For a balanced system, the networked store should fully use bandwidth offered by its network card, either 1 or 10GbE. Maps may use data spread across

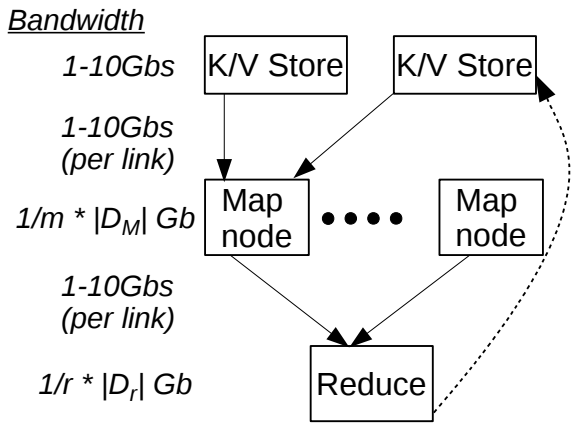

Figure 4.1: Data processing backed by networked storage under the map reduce model. Processing rate (bandwidth) at each stage is shown on the left.

multiple stores for three reasons. First, other unrelated jobs may lower the bandwidth available on a networked store [133]. Second, maps that access many small keys can encounter bottlenecks in TCP, operating system, and network congestion. Finally, networked stores that access disk have about 1/16th the bandwidth as 10GbE networks. Partitioning allows map jobs to regain lost bandwidth.

As shown in Figure 4.1, the map phase is often the slowest. Because of this, the map reduce model parallelizes this phase as much as possible. Let m be the average map time and $|D_M|$ be the average working set per map. If $|D_M|$ falls below 0.1Gb on a 1Gb/E network, then m must fall below 0.1 seconds to avoid slowing down the system. On the other hand, a map task that completes in constant time would require parallel data access as the data sizes grow. Reduce times are usually smaller than map times. They do not bound map reduce overall system times, and thus are not the focus of this chapter.

4.1.1 Outliers in Networked Storage

Networked storage is a more complicated storage fabric than local disks. Networked stores may include processors, DRAM, SSDs, and rotating disks. Operating systems and middleware connect these hardware. A mishap by any of these components can slow down access times by a significant amount. Networked stores are known to have outlier access times that are much slower than normal access times.

The root causes of outliers vary. For a concrete example, consider write buffering in a key-value store. To keep fast response times, most stores keep a relatively large in-memory write buffer. The buffer is flushed to disk periodically (every few seconds) to ensure a degree of fault tolerance to power loss. Writes that hit in the buffer can proceed at the speed of main memory, completing within a few hundred microseconds. However, writes that are stuck behind a buffer flush may be delayed by several hundred milliseconds. Other well-known root causes for buffer flushes include: OS scheduling background jobs, DNS timeouts, and garbage collection.

Figure 4.2 shows the access times for a Redis [122] store deployed on a 2GHz core with 2GB main memory. The workload shown represents 100% reads, and is tested under mean CPU utilization of 65.45% (high) and utilization of 10.75%(low). Under low utilization, we observe a heavy tail beginning with the 99th percentile, but when the Redis store is heavily utilized, we observe a heavy tail earlier, beginning with the 95th percentile. Most importantly, we note the respective lengths of these heavy tails. The exponential and Pareto distributions plotted here use the low utilization access times as a basis. The point marked on Figure 4.2 with a gray diamond marks the spot where the 99.99th percentile of the

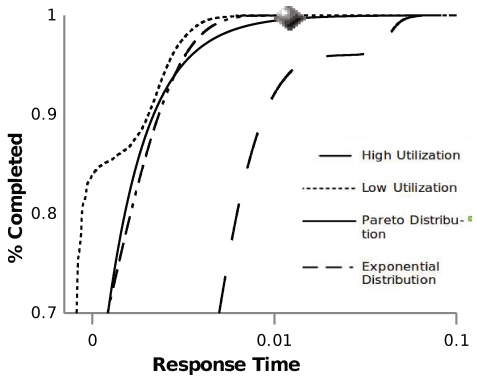

Figure 4.2: A cumulative distribution function regarding the times to access a Redis store under high and low utilization, shown with a Pareto distribution and an exponential distribution based on the low utilization numbers. The 99.99th percentile of the exponential distribution's heavy tail is marked.

exponential distribution occurs. The low utilization, high utilization, and Pareto heavy tails are much longer. The results are similar in production systems. Google BigTable reports default access times where the 99.9th percentile is 31X the mean [22]. Other works have noted similar results with MemCached [69].

4.1.2 Workloads that Reduce Big Data

Workloads that reduce big data to smaller chunks can use fast networked storage well. These workloads access a lot of data per map and they complete map tasks quickly. Graph analysis and data mining are well-known examples of such data reduction. Consider the problem of finding 2-hop friends in a social network. One approach pulls in data from a large subgraph of the network and then looks up all unique 2 hops within the subgraph from the origin friend. The subgraph itself can easily exceed $10MB$, yet looking up $16K$

hops during each map task can complete within milliseconds. Many data mining problems have similar properties due to statistical sampling.

Widely used tools for data processing, like Hadoop, target data transforms—not data reduction. Map tasks for transforming data take longer since every bit is touched. The Hadoop manual [151] calls for map jobs that take hours (meaning $|D_M|$ would need to exceed 36TB/s to balance network speeds). Workloads like Terasort provide such semantics. Emerging platforms for graph processing are more inline with big-data reduction [82, 49, 119].

4.2 Problem Statement

Figure 4.3 depicts a delay caused by an outlier access to networked storage in a data processing workload. Data accesses to Redis are pipelined, keeping all nodes busy in the ideal case. In the common case, the map node receives data just before it is needed. However, the last access on partition 0 is orders of magnitude slower than usual, preventing the next map from beginning. Such delays reflect lost throughput. Even if subsequent data accesses complete more quickly than usual, the pipelined nature of map tasks would not speed up.

This chapter explores two questions:

1. *How much do outliers slow down data processing?*
2. *Can we effectively mitigate outliers with redundancy?*

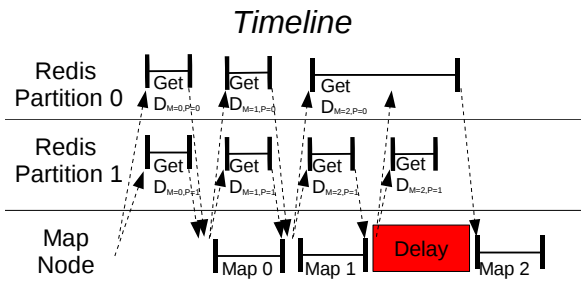

Figure 4.3: Slowdown caused by an outlier access to networked storage. Dotted lines are messages over the network. Solid lines reflect processing. For simplicity, we show all accesses for a single map stemming from a single network message.

4.3 Modelling Outliers

We used Operational Laws to model resource needs for networked stores and map nodes in a balanced system. We converted resource needs into expected delay. Finally, we used stochastic analysis to capture delays caused by outliers. This section describes our efforts. Table 4.1 describes the model parameters used in this section. We set controlled parameters directly. We restricted storage capacity per map node (C) to positive integers, meaning each map node pulled data from 1 or more dedicated storage partitions. A map node would pull from more than 1 storage partition in parallel if it needed access to a large working set. We varied map times (m) from 20ms (small) to 5s (large). We set the Pareto coefficient to control the heaviness of access time tails from the networked store: lightly heavy tail (1.76), normal heavy tail (1.44), and heavy heavy tail (1.13). In a nod to real system managers, we allow for some reserved, unused capacity (u). We set this parameter to 5% universally for networked stores.

	Controlled Model Inputs
C	Storage capacity per map node
m	Average map time
α	Pareto coefficient of the networked store
f	Reserved (unused) capacity on the networked store
	Derived Model Parameters
μ	Mean service time for the networked store
$\tilde{x}$	Median service time for the networked store
a	Average accesses per map
s_n	Slack time produced by n accesses
$\phi(t)$	Probability of an access longer than t

Table 4.1: Model Inputs.

We also made the following assumptions about our target systems:

- The networked store supports gets and puts on keys and values.
- The size of keys and values are fixed. In our tests, we use 1KB blocks.
- Maps know which data to request in advance before they execute.

We believe these assumptions can be relaxed in the future without changing our conclusions.

We used the control parameters and our assumptions to derive other parameters. First, we computed the average and median access times in a Pareto distribution given the Pareto coefficient (α).

$$\tilde{x} = X_{min} * \alpha^{0.5} \tag{4.1}$$

$$\mu = \frac{\alpha * X_{min}}{\alpha - 1} \tag{4.2}$$

Here, X_{min} is the smallest observed access time. We set this to 600µs based on data from Figure 4.2.

We used the Utilization Law to get the average accesses to storage per map. In a balanced system, the quotient of map time divided by average access time should equal average map time divided by reserve capacity. Accesses per storage partition per map simply divides this number by the storage capacity.

$$a = \frac{m}{\mu} * (1 - f) \tag{4.3}$$

$$a_i = \frac{a}{C} \tag{4.4}$$

Next, we computed *slack time*, the minimum delay for 1 outlier that could delay a map task. Slack time depends on the number of storage accesses that follow an outlier. An outlier followed by many accesses can be masked if subsequent accesses complete quickly. An outlier followed by only a few accesses is more likely to cause a delay. In our approach, slack time is comprised of two components. First, we turned the unused, reserved capacity (f) into idle time by multiplying this by the average map time. Then, we added the

difference of the mean and the median, multiplied by n. This means that an outlier that occurs when there are n outstanding accesses to the networked store can be masked if the remaining accesses complete according the median. For simplicity, our model makes the quantity of final storage accesses proportional to the over-provisioning range.

$$n = a * f \tag{4.5}$$

$$s_n = m * f + \frac{n}{C} * \mu - \frac{n}{C} * \tilde{x} \tag{4.6}$$

Given n, we can compute the probability and expected delay of an outlier that exceeds slack time. If networked stores had exponentially distributed access times, the expected delay would be fixed. However, heavy tail Pareto's are more complex. The first equation below computes the cumulative distribution function given α, X_{min}, and s_n. The equation after that computes the probability that 1 of n accesses is greater than s_n, i.e., the probability of a delayed map.

$$\phi(s_n) = 1 - \frac{X_{min}{}^{\alpha}}{s_n} \tag{4.7}$$

$$Pr(x > s_n) = 1 - \phi(s_n)^n \tag{4.8}$$

$$E(x|x > s_n) = \frac{2^{\frac{1}{\alpha}} X_{min}}{[1 - \phi(s_n)]^{\frac{1}{\alpha}}} \tag{4.9}$$

The final equation shows the typical (median) access time for such an outlier. The median delay of an outlier is the middle percentile starting from $\phi(s_n)$. A quick check reveals that when $\phi(s_n) = 0$, the result is the equation for the global median in a Pareto distribution.

Model Results: For Figure 4.4, we fixed storage capacity per node ($C = 4$), the Pareto coefficient, and unused capacity ($f = 5\%$). We controlled average map time and studied its effect on the slowdown caused by outliers. We show the equation for slowdown below:

$$slowdown = \frac{m + Pr(x > s_n)E(x|x > s_n)}{m} \tag{4.10}$$

We found that large map times (>5s) have first-order effects on slowdown. Large map times hide outliers in two ways. First, m is the only parameter in the denominator in our slowdown formula above. An outlier that causes the same absolute delay leads to less slowdown under large map times. Also, large map times can afford more slack time. Hadoop workloads often have large map times. In fact, the Hadoop manual calls for workloads with large (many minutes) map times [151]. Such workloads might disregard the impact of outliers when they move to networked storage.

On the other hand, data reduction workloads, e.g., graph processing, often have small map times. For example, a map may de-reference a few links in a large graph. Workloads with small map times suffer under heavy tail outliers. Our model expects that maps that take less than 100ms will be delayed (on average) by 5-15%. The heaviness of the tail also matters. Our heavy heavy tail setting caused up to 12X and 30X more slowdown than the normal heavy tail and light heavy tail.

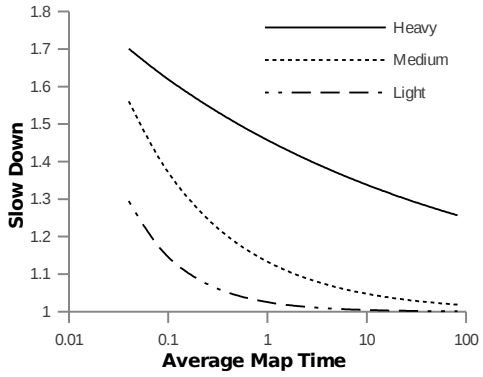

Figure 4.4: Slowdown caused by outliers as average map time varies.

Our model showed that slowdown is proportional to storage capacity per map node. Heavy tails cause outliers at a higher rate, but the effect remains linear. Even though the effects are only linear, reasonable ranges for I/O capacity lead to the largest slowdown. When each map node must contact 8 partitions in parallel, our model expects minimum slowdown around 40%. Even with just 4 nodes per partition, the worst case slowdown can exceed 63%.

4.4 Replication for Predictability

The model used in the previous section quantified the delay caused by outliers across map times and storage capacity. Outliers cause large delays for balanced systems with fast map times. Also, outliers cause large delays when the working set for maps is large. This section studies the potential for using replication for predictability as a solution.

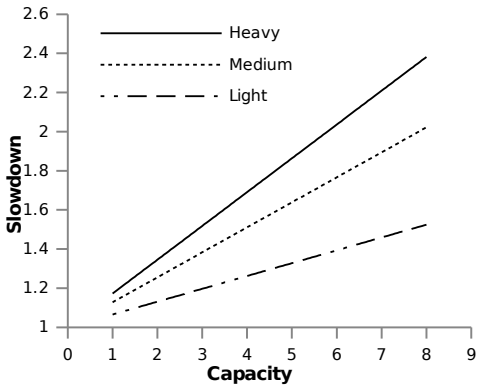

Figure 4.5: Slowdown caused by outliers as storage capacity per map node varies.

Replication for predictability is an old but seldom used technique to mask outliers that manifest independently. The basic idea is simple. Instead of sending storage accesses to only one node, send them to multiple nodes and use the result from the first node to respond. Intuitively, it is unlikely that all *duplicates* will return a slow result.

Replication for predictability has been rarely used in practice. Even though it reduces the effect of outliers, it does not improve throughput. Replication for predictability also does not reduce the effects of correlated outliers. For example, accesses to rarely viewed content will be slowed by cache misses in both redundant nodes. Thus, the key question for replication for predictability is, *can it be cost effective?*

To assess whether an idea is cost effective, we must model the cost and the return. We call the ratio of these terms the yield. In this chapter, we study a simple way to use replication for predictability sparingly. We use idle (unused) capacity on the networked store, i.e., u in Table 4.1. To be concrete, the cost is 5% of networked storage resources. For that investment, we hope to make the system more predictable and to recover throughput lost

to outlier effects. We measured yield as the return in slowdown divided by the investment. The full equation for yield is shown below.

$$yield = \frac{slowdown_{default} - slowdown_{rp}}{f} \tag{4.11}$$

Our model of replication for predictability assumed that storage accesses would be sent to only two duplicates. In ongoing work, we have extended the model to scale [170]. To capture the effects of replication for predictability, we have changed two aspects of the model presented in Section 4.3. First, accesses per storage node (a) ran at full capacity. Note, operating at full capacity increases the waiting time for accesses to networked storage. For interactive services, slow response times are costly. For the high throughput data processing workloads that we target, slow response times are only costly if they lead to delayed map jobs. In other words, our concern is the effect of queuing on slack time (s_n), where full capacity removes the buffer idle time. Updated equations are shown below.

$$a = \frac{m}{\mu} \tag{4.12}$$

$$s_n = \frac{n}{C} * \mu - \frac{n}{C} * \tilde{x} \tag{4.13}$$

On the positive side, replication for predictability reduces the chance of an outlier. If we assume that outliers arise independently, then the benefit of replication for predictability is shown below.

$$Pr(x > s_n) = 1 - \phi(s_n)^{2n} \tag{4.14}$$

Model Results For Figure 4.6, we again computed our model with all control parameters fixed except for the average map time. This plot shows the effect of map time on yield. We observed an effect that is comparable to the slowdown curve, but less dramatic. Map times below 100ms only reach yields ranging from 0.9–1.3. This result indicates that small map time alone do not warrant replication for predictability as this chapter proposes. For small map times, outliers beyond the last n may cause delays. Our sparing use of replication for predictability does not mask such outliers.

Looking deeper into Figure 4.6, the effect of outliers outside of the last n are most evident in the heavy heavy tail setting. At first, we expected this setting to provide the highest yield. However, looking further into the results showed that most of the delay under a very heavy tailed access time distribution was caused by accesses outside of the last n.

Figure 4.7 shows that high storage capacity per map node is sufficient reason to warrant replication for predictability. After capacity per node exceeds 5, we observed only high yields (>1).

Discussion: We have focused on the cost of replication for predictability in terms of storage access rate. The approach also uses network bandwidth. Since our model studied a limited use of replication for predictability, we did not consider this cost. If replication for predictability is expanded to use more resources, a topology aware approach may be needed [86].

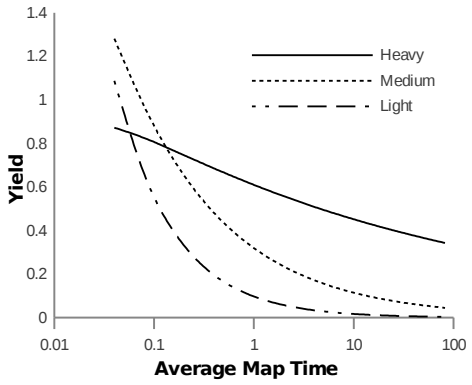

Figure 4.6: Yield caused by replication for predictability increases as average map time decreases.

Our model predicts yield but does not judge its value. Our intuition suggests that yield above 1 is a good investment, but ultimately, any novel scale out technique must be compared to other alternatives. If 5% spare capacity can be used in another way that provides higher yield, then replication for predictability should not be used.

Finally, we assume that the data processing platform comprises mostly reads. If writes were more frequent, we would need to consider consistency challenges posed by replication for predictability.

4.5 Related Work

Networked storage is a (re)emerging trend in high-throughput systems. However, networked storage is inherently more complicated than other storage mediums, e.g., disk. This chapter studies one product of such complexity: Heavy tailed access times. We make

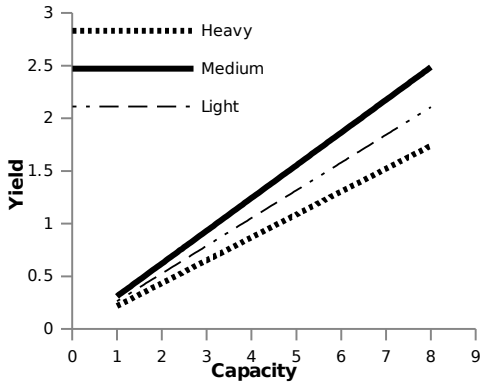

Figure 4.7: Yield caused by replication for predictability increases as storage capacity per map node increases.

the case for a research agenda that studies this phenomena. Prior research has 1) sped up networked stores or 2) improved overall throughput for data processing.

Speeding Up Networked Stores: MemCached and Redis are widely used open source networked stores [29, 122]. They both achieve high throughput (80K–100K requests per core). Other stores proposed by researchers have achieved high throughput also [69, 133, 90]. A common approach across these stores is to avoid touching disk, keeping operations within main memory. While key-value stores are most widely used, in-memory database systems have also gained traction. These databases relax their support for distributed transactions and also stay within main memory. When data sizes approach the capacity of main memory, it is better to compress data than to go to a single local disk [90]. Along with high throughput, stores can lower their latency by streamlining their execution path. [69] used soft direct memory access to remove the operating system for the data path for MemCached. These approaches make networked storage faster in the common case, however outliers (due to garbage collection, snapshots, etc.) still persist.

Balanced Data Processing: Disk is the primary component that has fallen behind. Recent work improves disk bandwidth by using multiple disks at each machine and modifying the data processing platform to access disk as little as possible [121, 120]. While these works have targeted data processing with node-local storage, they apply to networked storage with high bi-sectional bandwidth as well. Further, making the system more complex by adding multiple disks behind the networked store exacerbates outliers.

Chapter 5: Cache Provisioning for Interactive NLP Services

Unstructured, natural language (NL) corpora are large and growing fast. As of this writing, Twitter receives more than 300M tweets per day, a 2X increase over 2010 [126]. TripAdvisor holds over 100M user reviews, a 2X increase since 2011 [43]. Search engines, question-answer systems, and other big-data services process user queries against such data. To meet tight response time limits, these services cache data in the main memory of large clusters. For example, TripAdvisor uses a MemCache cluster on Amazon EC2, and this cluster comprises 52% of its online storage costs [43, 59]. As data grows, these costly caches require additional resources.

Given the costs of long response times, many services that process natural language data are designed to compute partial results quickly rather than full results slowly [55]. These services impose processing timeouts; a query that times out accesses only a fraction of its data. The difference between results returned with timeouts enabled (i.e., constrained resources) and results with infinite resources is *quality loss*. Users are often satisfied as long as quality loss is small. Large, in-memory caches prevent quality loss by allowing queries to access a lot of data within processing timeouts. However, NL corpora present a challenge: Documents contain redundant information. Services can over provision caches when the corpora grows faster than its informative content. Over provisioned caches inflate

operating costs by forcing managers to expand capacity sooner than needed. With memory prices dropping by an average of 30% per year [64], it is cost effective to wait as long as possible before buying resources.

This chapter argues that caches for NL workloads should be provisioned for quality loss, not data growth. These workloads permit some quality loss because NL concepts, e.g., synonyms and noisy results, introduce redundancy into query results. We present an approach to measure quality loss that captures these concepts. First, a query's baseline results were defined as those computed under a fully provisioned cache with no timeouts. We computed quality loss by comparing the baseline results with results observed under smaller caches. Queries had access to the same available data within a quality-loss test, but between tests we replayed data growth.

We set up two systems: Apache Lucene [153], an open source search engine, and OpenEphyra, an open source question answering system like IBM's Watson [34] that uses unstructured data. We used two NL datasets: Wikipedia and The New York Times. We organized each corpus into monthly snapshots, allowing us to measure quality loss over time as data grew. The portion of the Wikipedia corpus used grew by at most 30GB per month. The New York Times corpus added at most 88MB per month. From 2006–2008, our Wikipedia dataset exploded by more than 3X in raw size. We used a Redis [122] cluster as a main memory cache in our setup, and Google Trends to create a sequence of queries that were popular during periods studied. We replayed queries one-by-one under processing timeouts.

Quality loss varied based on 1) the corpus and 2) cache management policy. Cache under provisioning almost always caused quality loss, but often the effects were small. However,

if our search engine permitted some quality loss among the top K query results, it could provision 50% fewer cache resources on both Wikipedia and New York Times. We further observed that the New York Times corpus permitted a greater degree of cache under provisioning.

We also studied the impact of well known cache management policies. In *term-based LRU*, we stored only Lucene's inverted index in our main memory cache. When the cache was under provisioned, least recently used terms were swapped out of memory. This policy is widely used in search engines that provide pointers to content, rather than the actual content. In contrast, question-answer systems and online review engines often provide actual content. These workloads may prefer *content elision*, in which certain documents are elided from the indexes. Content elision is commonly used when new data replaces old data and the active size of the corpora is fixed. Terms in the resulting inverted index references fewer documents compared to inverted-index terms derived from the full corpora. In the worst case scenario of content elision, which we analyzed, new data is indexed only after a quality loss threshold is exceeded. Term-based LRU incurred less than 30% quality loss on both corpora. This result held even when the cache was severely under provisioned. Content elision incurred less than 30% quality loss on only the New York Times corpus. We hypothesize that content elision required more data redundancy to be effective.

We also used our framework to study the following policy: When quality loss exceeds a threshold, add more servers to expand the cache. We compared this approach to other approaches, including naively provisioning enough resources to fully provision the cache for the full corpus. Our approach reduced costs in two ways. First, it provisioned resources on demand, reducing operating costs. Second, it would enable managers to buy hardware

later rather than sooner, taking advantage of falling DRAM prices. Compared to buying enough memory servers upfront to handle 3 years of data growth, our approach reduced costs by 92%. Compared to an on-demand approach driven by monthly data growth, our quality-aware approach reduced costs by up to 48.8%.

The remainder of this chapter is organized follows: Section 5.1 defines quality loss in the context of NL workloads. Section 5.2 describes our experimental results. Section 5.3 discusses related work.

5.1 NLP Workloads

We interact with NL throughout our lives. We have learned to tolerate imprecise typographical errors, grammar, accents, and idioms. Services that process NL corpora also benefit from precision tolerance. We classify two key types of precision tolerance based on our experience with search engines.

Synonyms: Words and word sequences often have the same meaning within the context of a query. The precise output of a search engine with fully provisioned cache may output links to many of these synonyms. However, users are satisfied when a subset appears on their screen. For example, a Bing search for "Flowers in Washington State" returns results on florists, gardening, and the Coast Rhododendron (state flower). With a smaller cache, some of these results would be elided, but as long as the categories are represented (on the top answer page), many users will be satisfied.

Noise Tolerance: Continuing the example above, adjacent search results on the answer page represent different categories. Users are often willing to parse unrelated categories to find

the desired content. In other words, a certain degree of noisy results are okay as long as users can find good answers.

5.1.1 Defining Quality Loss

Quality loss (QL) is a metric to determine answer dissimilarity between an underprovisioned system and a fully provisioned baseline. To compute quality loss, we use the equation:

$$QL(x, \hat{x}, D, Q) = 1 - S(x, \hat{x}, D, Q)\ , \tag{5.1}$$

where x is our current underprovisioned hardware, software, data, and settings configuration, and $\hat{x}$ is the same configuration with enough cache resources to avoid timeouts. The function S is a measure of answer set similarity. We issue a set of queries Q and, for each query q_i, we compare its answers under x to its answers under $\hat{x}$.

Our similarity function is based on recall of the top-k results. We perform k-pairwise string comparisons, matchings top results under $\hat{x}$ (i.e., $R_i(\hat{x}, D)$) to results under x. When we find a match, we count it and move on to the next result string from the $\hat{x}$ answer set. At most one match for a single answer from a single question will be counted. The total number of matches is divided by K and averaged across all questions in Q. Equation 5.2 captures this base model and extends it to handle synonyms and noisy data.

$$S(x, \hat{x}, D, Q) = \frac{\sum_Q \sum_K \phi(\sum_{k_2} |R_{q,k}(\hat{x}, D) \cap R_{q,k_2}(x, D)|)}{|Q| \cdot K} \tag{5.2}$$

Capturing Synonyms: We specify a parameter K to use in a top-k analysis of quality, and thereby only look for matches of the first K result from the answer set. For example, in web search, K can be set as the number of results on the first page; these $K = 10$ results are the most critical to deciding result quality. As K decreases, the number of potential matches decreases and the denominator decreases; but as K increases, the difference between the current quality loss and the quality loss at $K - 1$ decreases until quality loss stays within 5% of the quality loss at the previous K.

Support for Noisy Results: Users are willing to look through some number of results to find what they were searching for. We add a parameter k_2 to capture this and revise the top-k analysis to top-2k analysis. Similar as top-k analysis, the top-2k analysis uses the K number of the top baseline results as denominator; but differently, it uses the number of matches between the top-K baseline results (from $\hat{x}$) and the top-k_2 test results (from x) as nominator, where $k_2 \geq K$. The relative difference of k_2 and K reflects users' tolerance level to noise. When users cannot tolerate any noise, we require $k_2 = K$; with higher tolerance of noise, k_2 can be more significantly larger than K.

Note, quality loss depends on the full specification of the above parameters and varies across services and users. A key contribution in our study is empirical analysis across a wide range of quality-loss settings.

5.2 Experimental Results

Figure 5.1 shows our system setup. For a given a query, Lucene's front-end nodes first look up query terms in a distributed Redis cache. Each Redis node stores up to 9 GB of data in

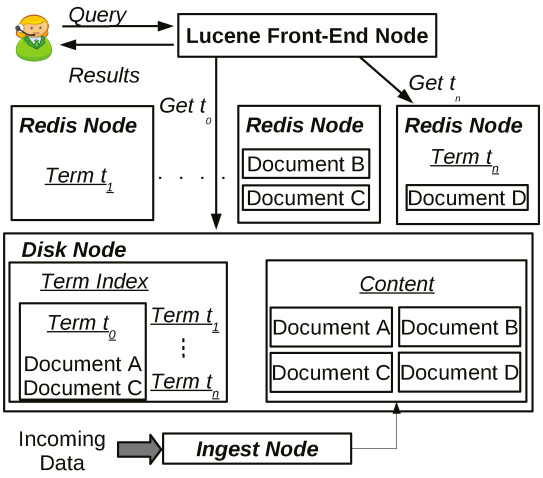

Figure 5.1: Our system setup for experimentation, including service logic for Lucene search engine and OpenEphyra question answering system.

its main memory. When more DRAM cache is needed, the cache scales out via additional Redis nodes. Terms not found in the Redis cache are looked up on two dedicated disk nodes that store 3 TB each. For each query, Lucene waits until all term data is found or a timeout occurs. Results are then analyzed, aggregated, and returned to the user. If the service logic in the application layer analyzes content, this content is also cached in Redis. For each experiment, we set query timeout, active Redis and disk nodes, and disk access times by broadcasting a configuration file to all nodes.

Our experiments run on a 112-node local cluster with EC2-like cloud provisioning. Each node has a 2.66GHz processor, 1 Gb Ethernet, and 100 GB local disk storage. Two dedicated disk nodes with the same specifications also have access to their own 3 TB external hard disk. The nodes described in Figure 5.1 communicate through software-defined image names.

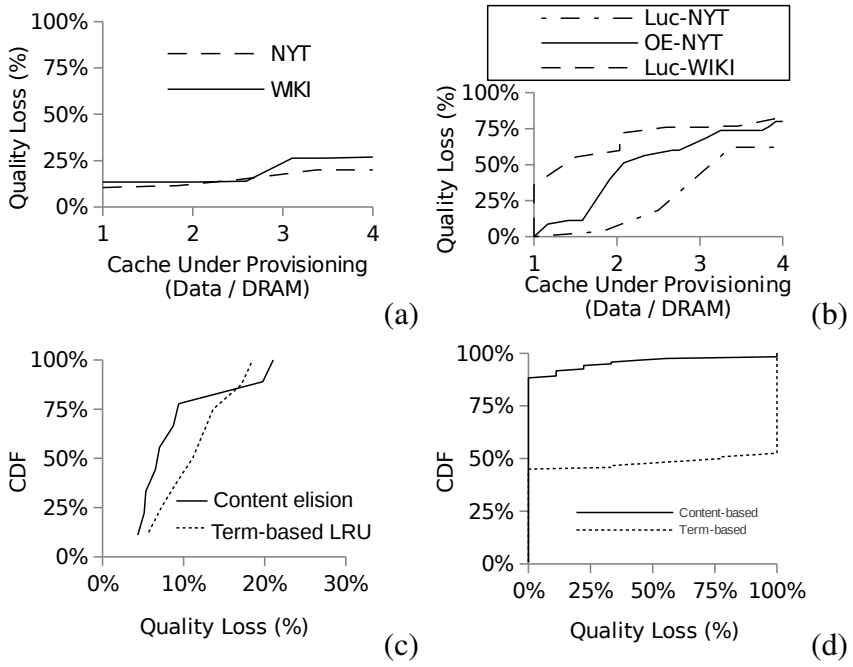

Figure 5.2: Cache under provisioning on quality loss. (a) Quality loss of NYT vs Wiki (b) Content elision caching (c) Distribution of quality loss by replacement policy (d) Quality loss per question.

We use Google Trends to capture 2,000 popular web searches representative of queries from 2004–2008. For each experiment, we replay these queries one by one. Typical response times are 500ms. We use Equation 5.1 to define quality loss. Our default configuration sets $K = 10$, $k_2 = 30$, and query timeout equal to 10 seconds.

5.2.1 Comparing NLP Datasets

Our data from The New York Times (NYT) spans articles published from 2004 to 2006. Over our trace, the corpus doubles in size to about 3GB indexed. However, new articles often repeat informative content from prior articles, reflecting follow-up stories and opinions pieces based on recent news articles.

Our data from Wikipedia (Wiki) spans articles published from 2001 to 2013, including revision data. We use two 3TB disk nodes to store the entire data set. Unlike New York Times, Wikipedia has less repeated content. Revisions often extend articles rather rephrasing existing content. However, links between entries can be repetitive, carrying over terms and copying definitions.

Figure 5.2(a) shows the observed quality loss across each data set as we increasingly under provision the cache, i.e., as data grew, we updated and increased the size of the term index on disk, but did not provision additional cache resources. Over time, the under provisioned cache pushed less popular terms to disk, increasing the probability that these terms would not be retrieved within the timeout. Both data sets handled under provisioned caches well; neither exceeded a 30% quality loss threshold.

5.2.2 Cache Replacement Policies

Figure 5.2(a) showed results where we updated the term index at each data snapshot. Under provisioned caches used LRU policies (a part of Redis) to manage growing data. For this section, we studied an alternative approach called content elision, in which the index size is kept static. In the worst case of content elision, the term index is not updated. Referring to Figure 5.1, we configured the ingest node to hold incoming data, instead of forwarding to the disk node. For services that store both term indexes and content in main memory caches, each new piece of data can use a lot of space. These services may prefer content elision because it prevents data growth. However, content elision can lead to high quality loss when incoming data is not highly redundant with existing data.

Figure 5.2(b) shows that quality loss under content elision varies depending on dataset and application. With Lucene on the NYT data, data growth can double the original cache size before hitting 10% quality loss. However, the less-redundant Wiki data suffers with quality loss starting at 35%. To analyze content elision, we set up an additional service, which accesses content and term indices stored within the Redis cache. OpenEphyra is question-answer system in the mold of IBM Watson [34]. It uses Lucene to identify documents related to a NL question and scans the top documents' contents for an answer. OpenEphyra uses the NYT workload. Our results show that for OpenEphyra up to 1.5 of the original data size can be added before hitting a 12% quality loss threshold. We suspect that the difference between OpenEphyra and Lucene on the NYT dataset is the effect of cache pressure from actual content access.

5.2.3 Whole Distribution Analysis

For Figure 5.2(c), we ran tests at a fixed under provisioning ratio (i.e., $\frac{data}{dram} = 2$). Each test used different data snapshots. We observe significant variance across the snapshots; quality loss increased by more than 3X for both caching policies. However, content elision has a significantly heavier tail relative to term-based LRU because when key documents are elided, the quality loss from content elision can affect many queries with terms described within the document. For Figure 5.2(d), we plot quality loss for one test on each question in our trace. Under content elision and NYT, we observe that most questions incur no quality loss at all, but the outliers that experience 100% loss (i.e., none of their results are the same) pull average quality loss up to 10%. Under term-based LRU with Wiki, we selected one of the worst data points (average quality loss was 45%) to highlight the on-off behavior of

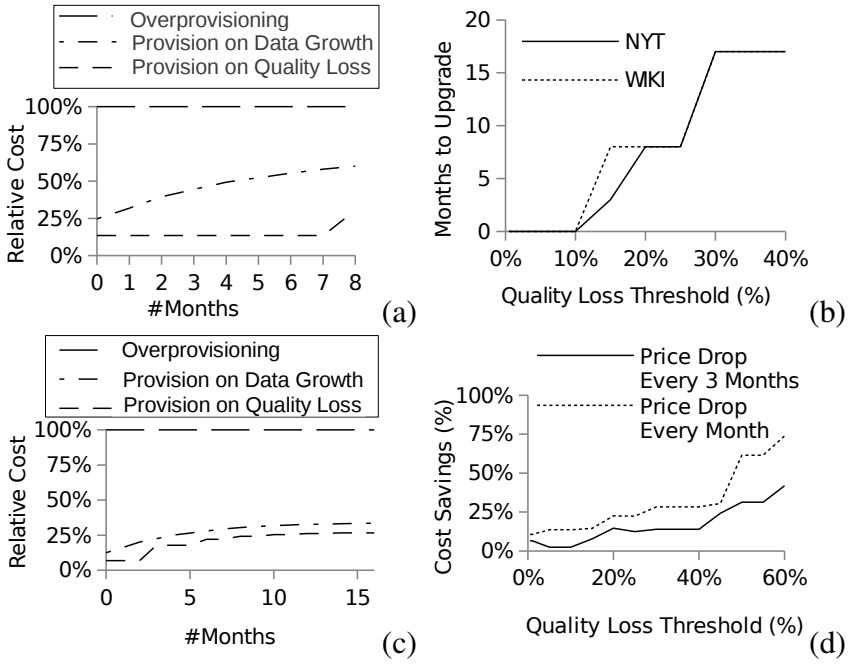

Figure 5.3: Cost savings of cache provisioning approaches. (a) Term-based LRU cache policy (b) Effect of quality loss threshold on term-based LRU (c) Content elision (d) Effect of quality loss threshold on content elision.

the replacement policy. If a query's terms are totally on disk, quality loss is high despite potential redundancy in the data.

5.2.4 Cache Provisioning on Quality Loss

In this section, we propose a new cache provisioning policy: *Expand the cache when when quality loss exceeds a threshold.* When quality loss does exceed a threshold, we add enough DRAM Redis nodes to fully provision the cache. Then we wait for quality loss to exceed the threshold again. By default, we set the threshold to 20%, but we explore the impact of all threshold settings. We call our approach *provision on quality loss*.

We compare our approach to two alternative provisioning policies. *Over provisioning* avoids any quality loss by provisioning enough resources to cache the entire NYT corpus up front. This policy has increased operating costs; since the average cost of DRAM is steadily decreasing, this policy also pays more per bit for cache resources. *Provision on data growth* provisions resources at each data snapshot, avoiding the increased price per bit from overprovisioning. As in our approach, this approach avoids the initial cash outlay. We assume all unspent cache budget is invested at 0.5% APR. Cost savings occur as interest gained from this investment plus the difference between the original price and the reduced price for DRAM.

We assume that DRAM prices drop on average by 2% per month, and simulate cost savings using our NYT and Wiki data for price drops every month and for price drops every three months. Figure 5.3(a) shows the cost of our approach relative to over provisioning and provisioning on data growth under the NYT dataset on Lucene, using term-based cache elision. Over 8 months, when our approach first provisions cache resources, our costs are 30% of the over provisioning case and half of the provisioning on data growth approach. Figure 5.3(b) shows the number of months that we can go before provisioning as a function of the quality loss. Here, we show results for both NYT and Wiki. For the New York Times data set under a DRAM price drop every three months, we save 19.45% compared to upgrading every time we add data. For the Wikipedia data set under a 3-month DRAM price drop, we save 14.37% compared to upgrading at every data add. For the New York Times data set, we save 51.19% compared to upgrading every month when we simulate a price drop every month; for the Wikipedia data set, we save 24.31% compared to upgrading at every data add when we simulate a price drop every month.

Figure 5.3(c) uses the same methodologies as the above but for content elision instead of term-based LRU elision. Our provisioning on quality loss approach saves more relative to the over provisioning approach, costing only 20%, but the provision on data growth approach is more competitive. This is because content elision requires updates to DRAM more frequently than term-based LRU. Figure 5.3(c) shows that when the DRAM cost drops every month, we save 22.51% of the cost of provisioning based on data growth and 80.44% of the cost of over provisioning.

As Figure 5.3(d) shows, the cost savings from increasing the quality loss threshold at an interest rate of 0.5% increases modestly when we we compare provisioning based on quality loss to provisioning based on data growth. With a quality loss threshold of 20%, we save 14.64% using the New York Times data set and 11.15% using the Wikipedia corpus for an every three month cost decrease. When the DRAM cost drops every month, we save 6.14% using Wikipedia and 22.51% using the New York Times dataset. This savings will grow as the threshold is relaxed; cost savings also increase as data is added. The numbers presented in this graph are subject to small fluctuations dependent upon the point at which we add data.

5.2.5 Additional issues

One of the parameters that affects quality loss regardless of caching policy used is the choice of presentation. All of the Lucene quality loss numbers presented in this chapter use the top-2k method of comparison, with a k of 10 and a k_2 of 30. As Figure 5.4 shows, the choice of k matters for the results coming from Lucene. A k less than 10 will result in

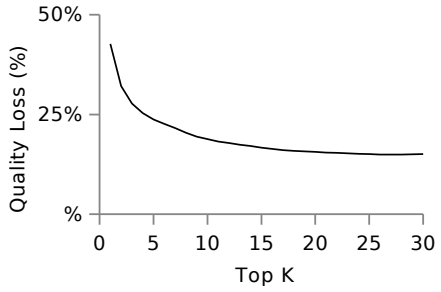

Figure 5.4: The effects of varying k on quality loss for a single experiment.

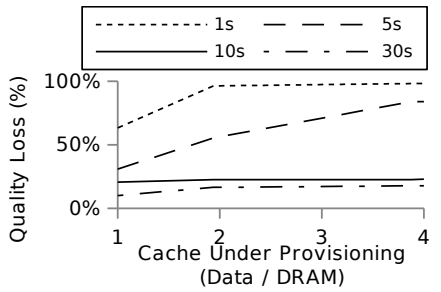

Figure 5.5: The effects of changing threshold on varied DRAM configurations over the same amount of New York Times data on Lucene using term-based caching.

showing higher quality loss than is average for the run, and a k greater than or equal to 10 will be result with quality loss within 5% of the average quality loss over all values of k.

Instead of changing the apportioned DRAM, we could modify the parameter that specified the timeout allowed by the system to analyze the effect of this timeout on quality loss.

Figure 5.5 shows the results of changing the timeout threshold over multiple different values of Data / DRAM. The lowest timeout threshold shown, at 1 second, resulted in a very high number of timeouts and very few results returned as compared to the other timeout thresholds shown. A timeout threshold of 5 seconds resulted in fewer timeouts and a correspondingly lower quality loss. A ten second 10 second threshold is slightly worse than the threshold with the lowest quality loss, which was 30 seconds.

5.3 Related Work

Our work intersects information retrieval, natural language processing, and storage systems. We exploit imprecision inherent in NL workloads to reduce caching costs. Our experiments with real NL workloads suggests that caches can be significantly under provisioned without incurring much quality loss.

Approximate computing also focuses on workloads that tolerate imprecision. For example, anytime algorithms [177] define a class of problems that can be solved incrementally. If the algorithm is interrupted during its execution, an imprecise result is returned. In contrast, compilers that support loop perforation [58] accept total running time as input. This approach elides loop iterations to complete within preset running times. Similarly, web content adaptation [38, 19] degrades image quality and webpage features to meet response time goals. Our own prior work [54] studies approximate computing within search engines, where a request may return partial results to complete within processing timeouts. These works, for the most part, trade off response time and imprecision. In contrast, our goal in this chapter is to trade imprecision for reduced cache costs. Baek and Chilimbi [10]

present a general framework to support approximated computation of different applications to tradeoff between quality and energy consumption.

Cache replacement and compression share our goal of provisioning fewer resources without incurring quality loss or high response times. SILT [90] is a key-value store that spans main memory, SSD, and disk. It combines diverse data structures across these materials, trading access time overhead with compression. Chockler et al. have begun studying caching as cloud service to achieve improved cache replacement under diverse, consolidated workloads [20].

Several recent works characterize application access patterns to reduce cache contention between competing applications [94, 143, 26]. Processing timeouts are akin to service level objectives. Recent work has shown that meeting strict objectives requires novel designs [61, 144, 23].

Capacity planners traditionally provision resources based on models of data growth, in part because non-NL workloads are less permissive to imprecision. Recent work from Google [146] models the growth of data. Their approach characterizes specific services and achieves predictably low error. Mackie [93] provides an earlier, macro-analysis forecasting approach.

Chapter 6: Obtaining and Managing Answer Quality for Online Data-Intensive Services

Online data-intensive (OLDI) services, such as product recommendation, sentiment analysis, question answering, and search engines power many popular websites and enterprise products. Like traditional Internet services, OLDI services must answer queries quickly. For example, Microsoft Bing's revenue would decrease by $316M if it answered search queries 500ms slower [37]. Similarly, IBM's DeepQA (deep question answering) implementation, Watson, would have lost to elite Jeopardy contestants if it waited too long to answer [87, 34]. However, OLDI and traditional services differ during query execution. Traditional services use structured databases to retrieve provably correct answers, but OLDI services use loosely structured or unstructured data. Extracting answers from loosely structured data can be complicated. Consider the OpenEphyra question answering system [130]. Each query execution reduces text documents to potentially relevant phrases by finding noun-verb answer templates within sentences.

OLDI services use large quantities of data to improve the quality of their answers. For example, IBM Watson parsed 4TB of data for its Jeopardy competition [34]. The amount of data used by these services is growing; Wikipedia alone grew 116X from 2004–2011 [148]. However, large data also increases processing demands. To keep response time low, OLDI

query executions are parallelized across distributed software components. These software components run in virtual machines distributed across cloud infrastructure. At Microsoft Bing, query execution invokes over 100–1000 components in parallel [66]. Each component contributes intermediate data that could improve answers. However, some query executions suffer from slow running components that take too long to complete. Since fast response time is essential, OLDI query executions cannot wait for slow components. Instead, they use anytime algorithms to compute answers with whatever data is available within response time constraints [177].

OLDI services use incremental computation over whatever data is available, returning the best available answers within response time constraints [54, 66, 98]. Parallel components can have different processing requirements based on skew in partitioned data, but a hardware failure or software anomaly could also cause severe performance degradation [8]. Returning these best available answers prevents slow parallel components from slowing down interactive queries. However, omitting data from slow components could degrade answer quality [123, 32]. In this chapter, answer quality is the similarity between answers produced online and without omitting data from slow components [72]. Queries achieve high answer quality when their execution does not suffer from slow components or when data omitted from slow components do not affect answers. Low answer quality means that omitted data from slow components have important contributions that would affect final answers significantly. Prior work has shown the virtue of adaptive resource management with regard to response time and quality of service [139, 41, 83]. Adaptive management could also help OLDI services manage answer quality. For example, admission control traditionally performs a check before accepting a query to determining if the system has resources available; in this context, admission control could check recent high priority queries for

low quality before admitting low priority queries to the queue. In this way, admission control could stabilize answer quality for high priority queries even under time-varying arrival rates by increasing shed of low priority queries when answer quality drops.

Answer quality is hard to measure online because it requires 2 query executions. Figure 6.1 depicts the process of computing answer quality. First, an online execution provides answers within response time constraints by omitting data from slow components. Second, we define a *mature execution* to use all available data relevant to a query by waiting for all components to complete before producing mature answers. Finally, a service-specific similarity function computes answer quality. This chapter uses true positive rate as the similarity function, but other functions are permissible, e.g., normalized discounted cumulative gain [95].

Prior research ran mature executions on dedicated offline testbeds [55, 70], but storage costs for offline testbeds grow as OLDI services ingest data. Further, a service's expected anwer quality per query depends on its query mix. Changing query mixes and data can affect answer quality online, which is difficult to measure in offline testbeds [70].

We present Ubora[1]*, a design approach to speed up mature executions.* Our key insight is that mature and online executions invoke many components with the same parameters. Memoization can speed up mature executions, i.e., a mature execution can complete faster by reusing data from its corresponding online execution instead of re-invoking components.

When a query arrives, Ubora conducts a normal online query execution except it records intermediate data provided by each software component, including data not reflected in the

[1]Ubora means *quality* in Swahili.

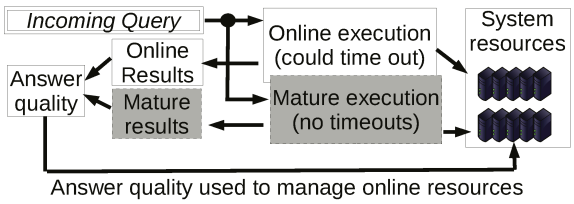

Figure 6.1: Steps to measure answer quality online. Mature and online executions may overlap.

online answers because they were omitted from slow components. After the slow components finish, Ubora computes *mature answers* using data recorded during and after the online execution. Implementing memoization for multi-component OLDI services presents systems challenges. First, OLDI services span multiple software components. It is challenging to coordinate mature and online executions across software components without changing application-level source code. Ubora manages mature and online operating context. During mature executions, it uses network redirection to replay intermediate data from in-memory storage. Second, memoization speeds up computationally intensive components but its increased bandwidth usage can also cause slowdown for some components. Ubora provides flexible settings for memoization, allowing each component to turn off memoization. We use offline profiling to determine which components benefit from memoization.

We have evaluated Ubora on 4 different open-source OLDI services with varying degrees of complexity and data size. To be clear, Ubora's systems-level implementation is able to support these applications without modification to their source code. We compared Ubora to query tagging, which changes application source code to resume mature execution at the point when an online execution returned a response. We also compared timeout

toggling, an approach which transparently applies the same context across all currently executing queries. Ubora completes mature executions nearly as quickly as query tagging with slowdown ranging from 8–16%. Ubora finishes mature executions 7X faster than timeout toggling. Finally, Ubora slows down normal, online query executions by less than 7%. We used Ubora to guide adaptive admission control. Ubora responded quickly to changing arrival rates, keeping answer quality above 90% during most of the trace.

This chapter is organized as follows. We describe the structure of OLDI services in Section 6.1. Section 6.2 explains the motivation for our work. We present Ubora in Section 6.3. Section 6.4 presents our implementation of query context tracking and tagging for memoization. In Section 6.5, we measure Ubora's performance using a wide range of OLDI benchmarks. In Section 6.6, we show that Ubora computes answer quality quickly enough to guide online admission control. In Section 6.7 we put our contributions in the context of related work.

6.1 Background on OLDI Services

Query executions differ fundamentally between online data-intensive (OLDI) and traditional Internet services. Traditional Internet services have query executions that process all data retrieved from well structured databases, often via SQL (i.e., LAMP services) [85]. Correct query executions produce answers with well defined structure, i.e., answers are provably right or wrong. In contrast, OLDI queries execute algorithms that increase in accuracy as time allows, including anytime algorithms [177]. They produce answers by discovering correlations within large quantities of data. OLDI services produce good answers if they process data relevant to query parameters.

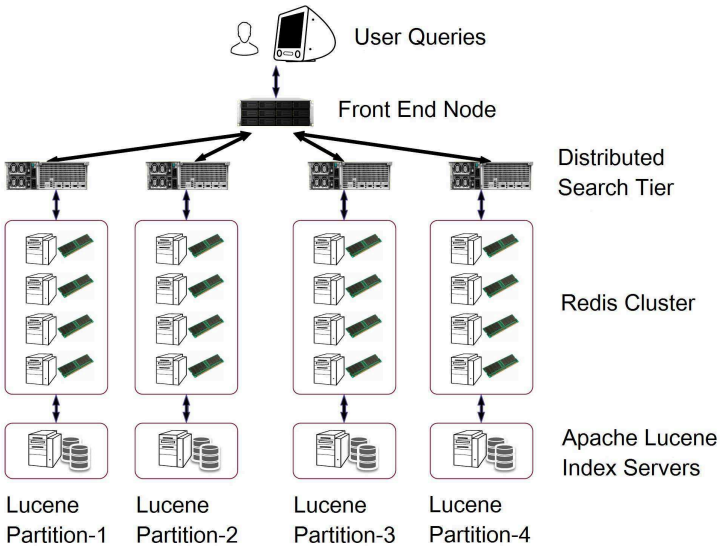

Figure 6.2: Execution of a single query in Apache Lucene. Adjacent paths reflect parallel execution across data partitions.

OLDI answers improve in quality with larger datasets. For example, IBM Watson competed at Jeopardy using 4TB of mostly public-domain data in distributed memory [34]. One of Watson's data sources, Wikipedia, grew 116X from 2004–2011 [148]. However, it is challenging to analyze an entire large dataset within strict response time limits. This section provides background on the software structure of OLDI services that enables the following:

1. Parallelized query executions for high throughput,
2. Returning best-effort, online answers based on partial data to prevent slow software components from delaying response time.

Parallelized Query Execution: Figure 6.2 depicts the query execution path in a service based on Apache Lucene, a widely used open-source information retrieval library [91]. This query execution path invokes 25 software components. Components in adjacent columns can execute in parallel. A front-end software component manages network connections with clients, sorts results from components running Distributed Search logic, and produces a running list of answers from results so far received. This list is returned to the user. Distributed Search software components parse the query, request a wide range of relevant data from partitioned storage components, and collect data returned within a given timeout. Data is retrieved from either 1) an in-memory Redis cluster that caches a subset of index entries and documents for a Lucene index server or 2) the Lucene index server itself, which stores the entire index and data on relatively slow disks.

The Lucene service in Figure 6.2 indexes 23.4 million Wikipedia and NY Times documents (pages + revisions) produced between 2001 and 2013. Queries access in parallel Lucene indexes partitioned across multiple virtual machines. A partition is a subset of data. Each parallel sub-execution (i.e., a vertical column) computes intermediate data based on its underlying partition. When intermediate data from parallel executions over each partition are combined, they compose a query response. This intermediate data is combined at a software component layer that may execute many queries in parallel, but processing for each query is done in sequential order (i. e., the front end node).

OLDI services also parallelize query executions via partial redundancy. In this approach, sub-executions compute intermediate data from overlapping partitions. The query execution weights answers based on the degree of overlap and aggregate data processing per partition. Consider a product recommendation service. Its query execution may spawn

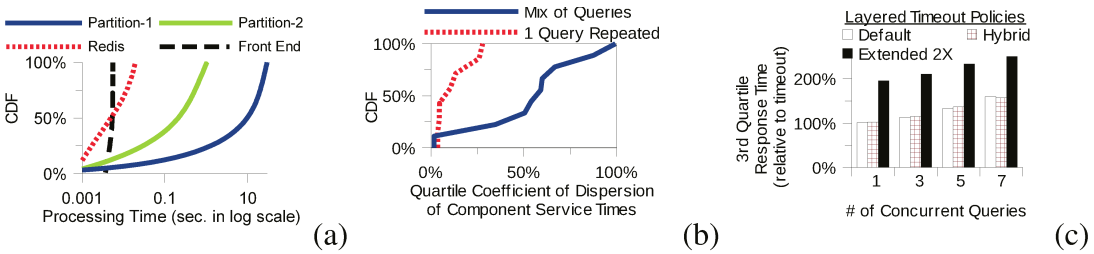

Figure 6.3: Experimental results with an Apache Lucene cluster. (a) OLDI components exhibit diverse processing times. (b) Query mix increases variability. (c) Timeout policies mask variation in favor of fast response times.

two parallel sub-executions. The first finds relevant products from orders completed in the last hour. The second considers the last 3 days. The service prefers the product recommended by the larger (3-day) sub-execution. However, if the preferred recommendation is unavailable or otherwise degraded, the results from the smaller parallel sub-execution help.

Online Answers Are Best Effort: In traditional Internet services, query execution invokes software components sequentially. The query response time depends on aggregate processing times of all components. In contrast, online data-intensive query executions invoke components in parallel. The processing time of the slowest component determines response time. Figure 6.3(a) quantifies component processing times in our Apache Lucene service. The query workload from Google Trends and hardware details are provided in Section 6.5. Processing times vary significantly from query to query. Note, the X-axis is shown in log scale. Lucene Index servers can take several seconds on some queries even though their typical processing times are much faster. Further, processing time is not uniform across partitions. For example, a query for "William Shakespeare" transferred 138KB from the partition 4 execution path but only 1KB from the partition 1 execution path. This is an

instance of data skew increasing the time to process partitions disproportionately. Partition 4 hosted more content related to this query even though the data was partitioned randomly.

6.2 Motivation

Extending Some Timeouts is not Enough to Achieve Mature Executions:

Many OLDI services prevent slow components from delaying response time by returning answers before slow components finish. Specifically, query executions trigger timeouts on slow components and produce answers that exclude some intermediate data. Timeouts effectively control response time. In our Apache system, we set a 2 second and a 4 second timeout in our front-end component. Average response time fell. Also, third quartile response times were consistently close to median times, showing that timeouts also reduced variance. Unfortunately, query executions that trigger timeouts use less data to compute answers. This degrades answer quality. For data-parallel queries answer quality degrades if the omitted data is relevant to query parameters.

Our Apache Lucene service answers mature queries too slowly to support interactive response times. First, since query mix changes often, the resource demands of parallel execution paths in OLDI services will vary significantly. Without excessive over provisioning, some queries will inevitably have paths that require more processing time than interactive services permit. To effect interactive response times, these paths will execute only for a preset time before halting, which causes answer quality to vary significantly. A second consequence arises because data growth affects data skew [3].

Impact of Timeout Policies To examine the impact of timeout policies on the maturity of online executions, we set a 3-second end-to-end timeout in our workload generator. The front-end software component timed out connections to the Distributed Search software components in 2 seconds. The Distributed Search software components timed out connections to Redis and Lucene index servers in 0.8 seconds. Such *layered timeouts* allow for some, bounded response-time variation from components but prevent large variation and failures from degrading end-to-end response time. We studied the impact of layered timeouts on the maturity of online executions. Figure 6.3(c) plots concurrency versus response time under this timeout policy. We compared the policy with default timeouts, doubled timeouts and a hybrid approach. This hybrid approach used a layered timeout, where two Lucene software components and a Redis software component had 10X timeouts but all others followed the default policy. Timeouts controlled response times well, triggering best effort results, which sped up some queries by 88X. Third quartile response times were consistently close to median times, showing that timeouts also reduced variance.

The hybrid approach is promising because it targets specific components and keeps response times low, but it fails to affect the results maturity. Even though some components had long timeouts, their executions were truncated by timeouts at higher layers. In contrast, extending all timeouts produced mature results but response time increased proportionally to the timeout extension.

Taken all together, our results show timeouts control response time for online executions effectively. Timeouts prematurely halt components affected by data skew, especially components replicated across data partitions. However, the exact components that trigger timeouts vary at runtime. Naively extending all timeouts may produce mature results, but this also increases average response time linearly. These lessons influenced our system design.

Variation of Mature Executions Given a query's parameters and data partitioning scheme, some components will be used more heavily than others. Data skew persists despite random hashing.

Processing times on partition 4 were 2.6X slower than partition 1. For example, queries related to the people that participated in the 2008 United States Presidential Election pulled much more heavily from Lucene partition 2 than Lucene partition 1, reflecting our partitioning strategy based on creation date.

We measure variation using the same query issued repeatedly and capturing the Quartile Coefficient of Dispersion ($QCOD = \frac{Q3-Q1}{Q3+Q1}$) for the response times of each component in our system. The Quartile Coefficient of Dispersion is a non-parametric metric like the coefficient of variation, but it is more robust to skew caused by outliers. Smaller numbers indicate that data is less spread out. Figure 6.3(b) shows that the quartile coefficient of variation increased signifcantly under a mix of queries compared to reissuing the same query. This illustrates that per-component processing times vary from query to query and across execution paths.

The variation of mature executions has practical consequences. First, since query mix changes often, the resource demands of parallel execution paths in OLDI services will vary significantly. Without excessive over provisioning, some queries will inevitably have paths that require more processing time than interactive services permit. Those paths will halt prematurely causing answer quality to vary significantly. A second consequence arises because data growth affects data skew [3]. Answer quality may differ significantly after an OLDI service ingests new data, even if query mix is stable. Both query mixes and data growth change online, so answer quality also changes online. Query mixes are non-stationary within 5-minute and 1-hour windows [140]. Data growth is ubiquitous. Facebook's Scuba ingests 1M events per minute [12]. TripAdvisor ingests 86K user reviews per hour [43]. These observations support our observation that answer quality changes dynamically.

We showed that response time variations across components and queries are present in modern OLDI services. These variations may be caused because of query-mixes and data skew. These results indicate that it is challenging to achieve mature results online by simply controlling a few components or workflow paths.

Ubora reduces resource needs by measuring answer quality for only randomly sampled queries, reusing computation from online executions and scheduling mature executions during low concurrency periods.

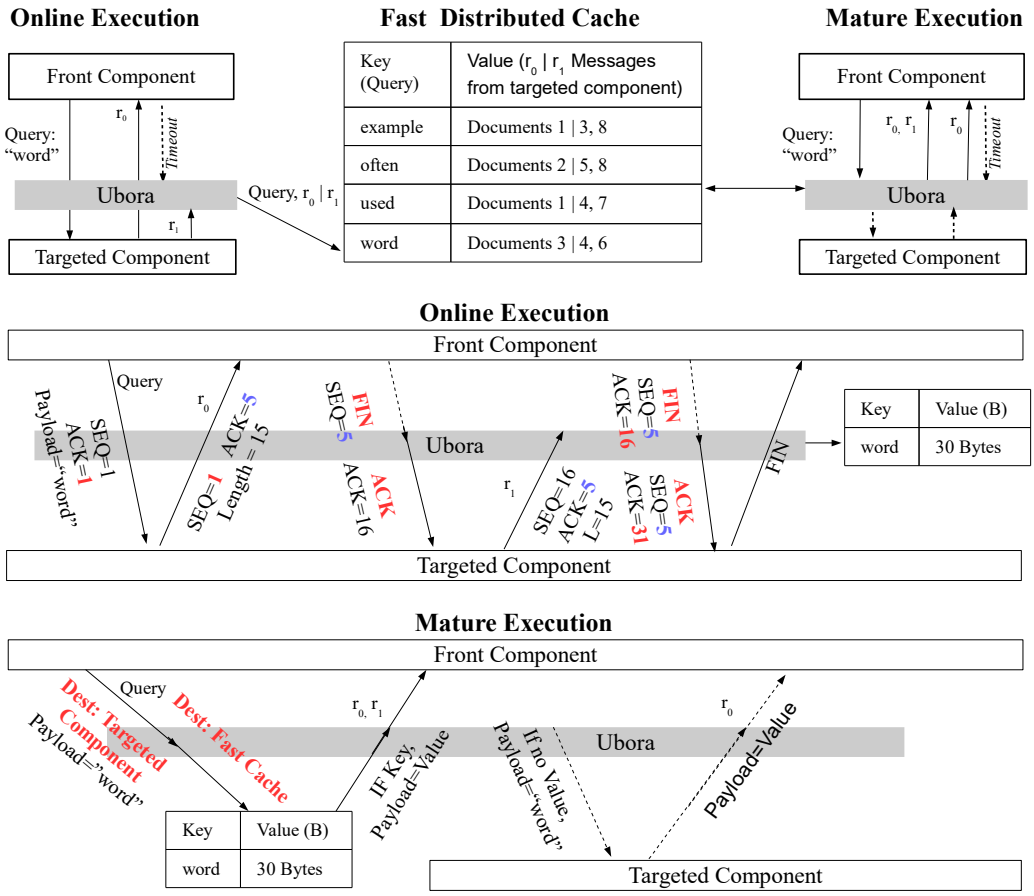

Figure 6.4: Memoization in Ubora. Arrows reflect messages in execution order (left to right). Dotted lines in Online Execution indicate communications that are transformed from their original purpose. Dotted lines in Mature Execution indicate communications that happen on occasion, as needed for correctness.

6.3 Design

By design, Ubora measures the answer quality of online query executions by comparing answers produced with and without timeouts. It reduces cost compared to other approaches by using existing online resources and employs memoization to speed up mature query executions. Memoization also reduces the overhead of executing mature queries online by

allowing reuse of previous intermediate results from targeted software components. Ubora further uses sampling to reduce overhead.

6.3.1 Design Goals

We designed Ubora around the following goals:

- Timeliness: The primary goal of Ubora is to measure answer quality quickly enough to enable resource management based on the results. For our purposes, the challenge was to acquire mature executions quickly, from an online environment. Because query mix changes over time, it is necessary to replay queries issued to the online service.
- Transparency: Require no code changes to software components. The secondary goal of Ubora is to support a broad range of online, data-intensive services composed of multiple different software components. For this to succeed, by design we require no changes to the code of software components used by the service. Instead of changing the services to require additional contextual information (i.e., online or mature), we use a middleware framework that tracks query context.
- Low Overhead: Online queries need to execute quickly, so we do not want slowdown. To this end, we introduce 2 optimizations which reduce slowdown in online queries. Sampling only a percentage of incoming online queries greatly reduces the overhead, as does delaying replay when needed to avoid queuing delay with online queries.
- Low Cost: While it is possible to compute mature results offline using an online query trace, this requires an increase in resources allotted to the service. In this section, we present an analysis of the cost of these additional resources to motivate why our design instead only uses resources currently available to the service.

6.3.2 Timeliness

Our design is mostly motivated by timeliness and transparency. We aim to overlap the mature execution as much as possible with the online execution. Once the online execution has completed, we keep targeted components executing in the background until they have fully processed the online query, and cache the results in distributed in-memory storage. When we then replay the online query, we use these cached results instead of accessing the targeted component.

Figure 6.4 depicts memoization in Ubora. During online query execution, Ubora records inter-component communication. It allows only front-end components to time out. Components invoked by parallel sub-executions complete in the background. As shown on the left side of Figure 6.4, without Ubora, this example front-end component invokes a component with a TCP payload containing a query, receives message r_0 and then times out. The front-end component then uses a FIN packet to trigger a timeout for the invoked component, stopping its execution before completion. A dotted line in Figure 6.4 shows Ubora blocking the trigger from the front-end component, allowing the invoked component to complete a mature execution. It records output messages before and *after* the front-end times out, in this case $r_0 + r_1$. These messages are cached in fast, in-memory storage distributed across the least utilized machines.

With Ubora, front-end components still answer online queries within strict response time limits. As shown in Figure 6.4, the front-end component uses r_0 to produce an *online answer*. After all sub-executions for a query complete, Ubora re-executes the front-end, as if a new query arrived. However, during this mature execution, Ubora intercepts messages to

other components and serves response messages r_0 and r_1 from cache (i.e., memoization). The cache delivers messages with minimal processing or disk delays. During this mature execution, the front-end uses both $r_0 + r_1$ to produce a *mature answer*. For correctness in mature executions, Ubora may connect to the targeted component if data is not in fast cache. This connection is shown as dotted lines in Figure 6.4.

Mature and online results are stored in cache. After replay completes, answer quality is computed by callback functions or scheduled jobs. Each service may define answer quality differently by providing its own function. Example functions include weighted top K, normalized discounted cumulative gain or true positive rate. In our experience, these functions complete quickly relative to query execution time.

6.3.3 Transparency

Queries execute under different contexts tracked by Ubora, depending on whether they are online executions being *recorded*, *normal* online executions, or mature executions being *replayed*. Such *execution context* requires coordination across distributed nodes and concurrent queries. Memoization should be implemented differently depending on available systems support for execution context tracking. Further, replayed executions may steal resources from high priority online executions. The challenge is to minimize queuing interference.

6.3.4 Low Overhead

Queries issued by users or other external sources must complete quickly. For sampled queries, a query's online execution completes during record mode. Record mode adds light overhead by sending messages to the cache, but importantly, most components execute under normal timeout settings. This approach is similar to the *hybrid approach* in Section 6.2. The query completes quickly due to layered timeouts while mature component-level executions happen asynchronously. In contrast, naively extending all timeouts increases response times. We introduce 2 optimizations to keep overhead low.

Replay Mode Can Be Postponed to Reduce Interference Replays can be temporarily postponed to avoid queuing delays with online executions, but they must finish in a timely fashion to impact online management. Fortunately, components operate under normal timeout settings during replay mode. Replay mode completes quickly and predictably.

Sampling Frequency versus Slowdown and Representativeness Ubora uses sampling to reduce the aggregate overhead of mature executions. Recall, mature executions use a lot more resources than online executions. Services do not have enough idle resources to complete a mature execution for every online execution. However, sampling too infrequently can inhibit online management because answer quality is produced too late or is not representative. The sampling rate allows managers to trade throughput on mature executions for processing overhead.

Our design allows each service to set a sampling rate that matches its hardware and query mixes. Online executions selected for record and replay suffer longer service times but unsampled queries execute normally.

6.3.5 Low Cost

While it would be possible to compute mature executions on an offline testbed using the online queries, this is a costly proposition, requiring 100% additional infrastructure cost in the worst case. To keep the cost of measuring answer quality low, we opt to share the resources already allocated to the service.

Memoization and replay modes redirect network traffic from nodes used to process online queries. The approach does not require an offline testbed. Offline testbeds require costly data replication for accurate mature executions. As datasets grow, testbed costs grow too. Next we show that offline testbeds are expensive by analyzing the growth of Wikipedia data.

Figure 6.5 shows the expected cost of our Apache Lucene setup hosting the Wikipedia data. We studied the dataset from 2004 to 2009. In those years, the dataset size (d_i) grew by 130%, 150%, 110%, 70%, 40% and 23% respectively, ending at 4TB [148]. Hard drive capacity (h_i) grew at 20% annually, starting at 250MB. We used Amazon EC2 pricing for reserved nodes ($2,400/year) and EBS storage pricing ($1.20 GB/year). We set annual inflation to 2%. The number of partitions is captured by Equation 6.1. The cost for online resources, i.e., Lucene partitions (p_i), 4 cache instances per partition and 1 distributed search components per partition, are captured by Equation 6.2. The least expensive cost of

an offline testbed that fully replicates data is modelled in Equation 6.3. It adds additional instances and doubles EBS costs but does not include offline resources for processing. A full offline testbed with 1 distributed search component and 1 cache cluster per partition is modelled by Equation 6.4.

$$p_i = \frac{d_i}{h_i} \tag{6.1}$$

$$C_{Online} = (6p \times \$2400) \times 1.02^i + (d_i \times \$1.2) \times 1.02^i \tag{6.2}$$

$$C_{Data} = (7p \times \$2400) \times 1.02^i + (2 \times d_i \times \$1.2) \times 1.02^i \tag{6.3}$$

$$C_{Offline} = (9p \times \$2400) \times 1.02^i + (2 \times d_i \times \$1.2) \times 1.02^i \tag{6.4}$$

Figure 6.5 depicts the cost of data growth and offline testbeds. Data growth alone would increase costs for our Apache Lucene setup by 3X between 2005 to 2009. We note that Wikipedia's reported hosting costs grew by 4.3X during the same period. A full offline testbed would increase costs by 50%. An offline dataset alone would increase costs by 18%. Further with inflation, the relative cost of an offline dataset compared to the cost of online nodes would grow from 18% in 2004 to 22% in 2009. Record, cache and replay reduces operating costs by 35% compared to approaches that maintain a full offline testbed.

Prior work proposed subsampling data to reduce costs [70]. A sampled dataset can only approximate the answer quality of online results. Still, an offline testbed that samples 20% of data would increase costs by $46,000 during the period studied.

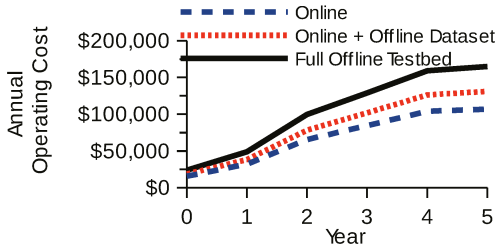

Figure 6.5: Annual operating costs for Apache Lucene on EC2 with Wikipedia growth rates.

6.3.6 Limitations

Timeliness of results and low overhead required that our solution occur in an online environment. Transparency is the requirement that motivates our approach as a networking solution. While a new kernel module would also have sufficed for the other requirements, we wanted for our solution to be usable out of the box for many services and systems. We aim that every service that communicates between components using a networking connection can find use in our design.

Our design approach is not fully automatic. System managers choose components to target. Identifying which components to target automatically would be a potential problem to solve in the future. Also, record mode assumes the targeted component responds to the component which invoked it [138]. In graph and event processing systems, the targeted component forwards data to the next node in the dataflow [106, 159]. Third, we assume that request execution is read-only or otherwise idempotent, because workloads that allow writes may have incorrect output on replay. Finally, cached output can be used to replay only one component in the parallel execution path of each query. Specifically, our approach

produces inaccurate results when two or more components that execute in the same sequential execution path time out, because replay can speed up only one. Section 6.2 suggests that picking components that interact with data partitions and are most affected by data skew suffices in many cases.

6.4 Implementation

This section discusses the implementation of Ubora. First, we describe axiomatic choices, e.g., the user interface, target users and prerequisite infrastructure. Second, we discuss how operating system support for transparent context tracking impacted the implementation of memoization. Third, we provide details about our implementation and optimizations made to keep overhead low. Finally, we discuss our approach to determine which components constitute a front-end.

6.4.1 Interface and Users

Ubora is designed for use by system managers. It runs on a cluster of compute nodes. Each node runs a networked operating system with many virtual machines. Each virtual machine runs 1 software component on its associated resources. To be clear, a software component is a running binary that accepts invocations over the network. Each software component has a unique network addresses, e.g., IP address and TCP port, assigned through its virtual machine. A cluster of nodes may run one or more services. Each service comprises a set of software components logically arranged in query execution flow paths, yet physically distributed across 1 or more nodes.

System managers understand the query execution paths in their service (e.g., as depicted in Figure 6.2). They classify each component as front- or back-end. Front components receive queries, record inter-component messages and produce online and mature answers. They are re-executed to get mature answers. Back-end components propagate query context, record messages, and do not time out for sampled queries. Figure 6.2 labels the front-end component. The search tier, Redis and/or Lucene could be front-end or back-end components.

Ubora is started from the command line. Two shell scripts, *startOnBack and startOnFront*, are run from a front component. Managers can configure a number of parameters before starting Ubora, shown in Listing 6.6. The query sampling rate is given in terms of the number of mature executions to initiate per unit time. When new queries arrive at front end TCP ports, a query sampler randomly decides how to execute the query. Sampled queries are executed under the *record mode* context shown on the left side of Figure 6.4. Queries not sampled are executed normally without intervention from Ubora. Record timeout duration sets the upper bound on processing time for a back-end component's mature execution. Propagate timeout is used to set the upper bound on time to scan for newly contacted components to propagate the execution context. To get mature answers the query execution context is called *replay mode*. Finally, the callback function used to compute answer quality is service specific. The default is True Positive Rate.

6.4.2 Transparent Context Tracking

A key implementation goal was to make Ubora as transparent as possible. Here, transparent means that 1) it should work with existing middleware and operating systems without

changing them and 2) it should have small effects on response times for online queries. Transparency is hard to achieve, because Ubora must manage record and replay modes without changing the interaction between software components. In other words, the execution context of a query must pass between components unobtrusively. Some operating systems already support execution contexts. Therefore, we present two designs. The first design targets these operating systems. The second design targets commodity operating systems. Our designs exploit the following features of memoization:

1. *Queries produce valid output under record, replay, and normal modes.* This property is achieved by maintaining a shadow connection to the invoked component during replay. Cache misses trigger direct communication with invoked components. As a result, replay, normal, and record modes have access to full data.
2. *Back-end components use more resources during record mode than they use during normal online execution because timeouts are disabled.*
3. *Replay mode produces mature results within normal online timeout settings, since the output of invoked components are replayed from fast cache.* Our design schedules replay executions to avoid queuing delay.

Transparency using OS Managed Contexts: Some operating systems track execution context by annotating network messages and thread-local memory with context and ID. Dapper [134] instruments Google's threading libraries, Power Containers [132] tracks context switches between Linux processes and annotates TCP messages and Xtrace [35] instruments networked middleware.

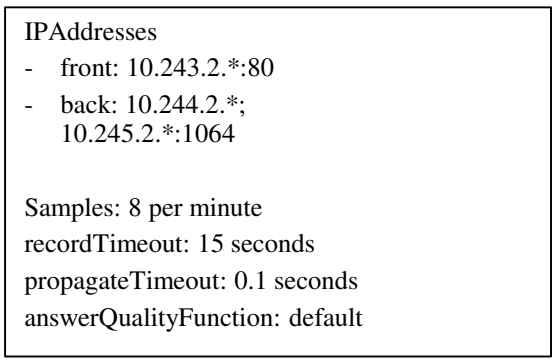

Figure 6.6: Ubora's YAML Configuration

OS-managed execution context simplifies our design. Ubora intercepts messages between components, acting as a middle box. Before delivering messages that initiate remote procedures, Ubora checks query ID and context and configures memoization-related context (i.e., record or replay mode). The same checks are performed on context switches. During record mode, when a component initiates a remote invocation, we use the message and query id as a key in the cache. Subsequent component interactions comprise the value associated with the key—provided the query context and ID are matched. We split the value and form a new key when the invoking component sends another message. Subsequent messages from the target would provide values for the new key.

In replay mode, when an invocation message is intercepted, the message is used to look up values in the cache. On hits, the cache returns all values that are associated with the message. The cache results are turned into properly formatted messages (e.g., TCP packets) to transparently provide the illusion of RPC. On misses, the message is delivered to the destination component as described above.

Transparency without OS Support: Most vanilla operating systems do not track execution context. Without such support, it is challenging to distinguish remote procedure calls between concurrent queries. A universal feature of online, data-intensive services is the use of distributed communication between software components. While other context tracking solutions are available, this network traffic is sufficient to allow transparency with regard to both workload and underlying operating system. Ubora's memoization permits imperfect context management because record, replay and normal modes yield valid output. This feature allows us to execute concurrent queries under the same context, but we still must ensure correctness. First, we describe a simple but broken idea that is highly transparent, and then we present an empirical insight that allows us to improve this design without sacrificing transparency.

In this simple idea, each component manages its current, global execution context that is applied to all concurrent queries. Also, it manages a context id that distinguishes concurrent record contexts. Ubora intercepts messages between components. When a component initiates a remote invocation in record mode, the message and context id are used to create a key. For the duration of record mode, inter-component messages are recorded as values for the key. If the context indicates replay mode, the message and context id are used to retrieve values from cache.

This simple idea is broken because all messages from the invoked component are recorded and cached, including concurrent messages from different queries. In replay mode, those messages can cause wrong output. Our key insight is that record mode should use replies from the invoked component only if they are from the same TCP connection as the initiating TCP connection. The approach works well as long as TCP connections are not shared

by concurrent queries. Widely used paradigms like TCP connection pooling and thread pooling are ideal for use with Ubora. We studied the source code of 15 widely used open source components including: JBoss, LDAP, Terracotta, Thrift and Apache Solr. Only 2 (13%) of these platforms multiplexed concurrent queries across the same connection. This suggests that our transparent design can be applied across a wide range of services. We confirm this in Section 6.5.4.

Next we describe how to propagate request context, which is necessary when the operating system does not support execution contexts. On a front component, the Ubora controller waits for queries to arrive on a designated TCP port. If a query is selected for mature execution, the Ubora controller changes the front component context from normal to record and create a context id. Before sending any TCP message, we extract the destination component. If the destination has not been contacted since record mode was set, the Ubora controller sends a UDP message to tell the Ubora daemon running on that component to enter record mode and forwards the proper context id. Then we send the original message. Note, UDP messages can fail or arrive out of order. This causes the mature execution to fail. However, we accept lower throughput (i.e., mature executions per query) when this happens to avoid increased latency from TCP roundtrips. Middle components propagate state in the same way. Each component maintains its own local timers. After a propagation timeout is reached, the context id is not forwarded anymore. After the record timeout is reached, each component reverts back to normal mode independently. We require front components to wait slightly longer than record timeout to ensure the system has returned to normal.

6.4.3 Prototype

We implemented transparent context tracking as described above for the Linux 3.1 operating system. The implementation is installed as user-level package written in C and requires the Linux Netfilter library to intercept and reroute TCP messages. It uses IPQueue to trigger context management processes. It assumes components communicate through remote procedure calls (RPC) implemented on TCP and that an IP address and TCP port uniquely identify each component. It also assumes timeouts are triggered by the RPC caller externally—not internally by the called component.

Ubora also implements a user-level controller that changes a node's operating mode to record, replay or normal. A single-writer but globally readable file holds the current operating mode on each node. In normal mode, rules regarding targeted components and query detection are disabled.

Recording Network Payloads: Our approach records messages sent from targeted components during live execution. First, headers are matched against rules about 1) identifying targeted components, 2) new queries and 3) Ubora control. Packets that do not match these rules are accepted (more precisely, they are not redirected). Second, we keep TCP connections to targeted components open after timeouts to obtain messages excluded from premature results. Ubora extends timeouts transparently by blocking FIN packets sent to the targeted component and spoofing ACKs from the caller. Messages from the targeted component that arrive after a blocked FIN are cached but not delivered to the caller. This continues until record times out or the targeted component sends a message to end the connection. Then, the entire recorded payload from the targeted component is stored as a value

in Ubora's cache, with the query payload from the caller that invoked this output as its key. While we delineate between key-value pairs by default using either FIN packets or the next inbound message to a component, service managers can specify alternate, application-specific data payloads to use for this purpose. This is especially important for workloads that use connection pooling with collated requests separated by special characters. We then separate messages using these application-specific termination payloads. Service managers can specify this in the configuration file. Packets are parsed in two stages.

Distributed Cache: Naively limiting our storage usage to a single node increases average response time by increasing that storage node's network bandwidth and decreasing the amount of data used by the host application that can be stored. Instead, we allow system administrators to volunteer a list of nodes on which to store recorded message pairs. Recorded message pairs can be matched to nodes by a random hash, decreasing network bandwidth for bandwidth intensive applications.

We use a distributed Redis cache for in-memory key value storage. Redis allows us to set a maximum memory footprint per node. The aggregate memory across all nodes must exceed the footprint of a query. By using only a small percentage of cache on each of allowed nodes, we minimize the overall cache miss rate overhead. Our default setting is an aggregate 1 GB. Also, Redis can run as a user-level process even if another Redis instance runs concurrently, providing high transparency.

We want to minimize the overhead in terms of response time and cache miss rate. Each key value pair expires after a set amount of time. Assuming a set request rate, cache capacity will stabilize over time. A small amount of state is kept in local in-memory storage on the

Ubora control unit node (a front node). Such state includes sampled queries, online and mature results and answer quality computations.

Replay: When we rerun the cached query, we spoof the targeted components using recorded messages. Replayed queries bypass processing and I/O delays on targeted components, returning results at network speed. If a packet is headed for the targeted component, the packet is intercepted, and Ubora accesses its cache for the requested key value pair associated with the data payload. The value is sent back to the source address, if the key was found in cache. If nothing was found, the data payload is sent to the targeted component for correctness. For correctness, when a key is not found in Ubora cache or the Ubora cache develops a fault, we also open a real connection to the targeted component and send the data payload.

Ubora's memoization approach meets the criteria outlined earlier. Other designs that we studied do not. For example, a system could measure answer quality by replaying queries on an offline testbed. This would violate our goal of using only online resources [70]. A system could measure answer quality online by rewriting system components to support query-specific timeouts as in Bing [66, 54]. However, this would require re-coding software platforms. Finally, a system could simply change timeouts for each component dynamically. However, layered timeouts would force such changes to cover whole workflows, i.e., no incremental deployment. This would hurt response times for all live queries.

6.4.4 Optimizations for Low Overhead

Context Tracking: Ubora reduces bandwidth required for context propagation. First, Ubora propagates context only to components used during online execution. Section 6.5.4 demonstrates that the increase in packets touched is more than compensated for in the decrease of Ubora network traffic during mode changes.

Second, Ubora does not use bandwidth to return components to normal mode, only sending UDP messages when necessary to enable record or replay mode. The naive context propagation sends messages for all context switches, including return to normal mode. Timeouts local to each component ensure that the system fully returns to normal mode, regardless of any lost UDP messages. Timeouts therefore increase robustness to network partitions and congestion. Front components time out after other components so that the entire system will have returned to normal mode before another mode change is issued.

Reducing Replay Overhead: To keep response times low, online services underutilize systems resources [97, 98]. Replay executions increase utilization on middle components. After record completes, Ubora queues the context id on the front component for replay. The expected time to complete queued replays is the product of queue size and average online execution time. Naively, Ubora replays queries for mature execution as soon as possible after the record mode completes.

However, we have reduced replay overhead by using three further factors to decide when to replay queries. First, replay queries must execute within a short window after online queries finish to be useful to online management. This expiration window is set by system

managers. If the time to clear the queue exceeds the remaining expiration window, Ubora initiates replays. Otherwise, replays are initiated if 1) there are no outstanding online queries and 2) the average inter-arrival time for online queries exceeds the time to replay. If these conditions are met, we replay the first query in the queue.

Second, by caching results without expiration, Ubora can run replay executions over a window of time after initial live executions. The value of delaying replay is that replay can be done when queuing delay is low, reducing the impact of replay on response times of other live executions. If services have frequent idle periods between queries, Ubora can schedule replays during such time. [97] found that such services idle about 70% but for less than 5 milliseconds at a time. Ubora can be set to read queue lengths at front end nodes and schedule replays when queue length is below the 25th percentile.

Sampling: Mature results do not directly contribute to end user satisfaction. Naively collecting mature results for each query would reduce an OLDI service's throughput by more than 50%. Ubora allows managers to specify stochastic sampling rates to determine when to compute a mature result.

Second, we use a node sampling optimization for applications with intensive data reuse. When this is used, recorded message pairs are stored on the node with the lowest Ubora storage footprint.

6.4.5 Determining Front-End Components

Thus far, we have described the front-end as the software component at which queries initiate. Its internal timeout ensures fast response time, even as components that it invokes

continue to execute in the background. To produce an online answer, the front-end must complete its execution. Ubora re-executes the front-end to get mature answers. Ubora can not apply memoization to the front-end component.

At first glance, re-execution seems slower than memoization. However, as shown in Figure 6.3, many components execute quickly. In some cases, execution is faster than transferring intermediate data to the key-value store. Our implementation allows for a third class of component: middle components. Like front-end components, middle components are allowed to time out. They are re-executed in replay mode without memoization. Unlike front-end components, middle components do not initiate queries, but they can invoke targeted components and they can be the target of memoization. In Figure 6.2, Distributed Search or Redis components could be labeled middle components.

Given a trace of representative queries, Ubora determines which components to memoize by systematically measuring throughput with different combinations of front-, middle- and back-end components. We do the same to determine the best sampling rate.

6.5 Experimental Evaluation

In this section, we compare Ubora to alternative designs and implementations across a wide range of OLDI workloads. First, we discuss the chosen metrics of merit. Then, we describe the competing designs and implementations. Then, we present the software and hardware architecture for the OLDI services used. Finally, we present experimental results.

Code-name	Platform	Parallel Paths	Data (GB)	Nodes	Maturity	Utilization	QCoD
YN.bdb	Apache Yarn	2	1	8	96%	46%	8%
LC.news	Lucene	1	4	4	82%	73%	13%
LC.wik	Lucene	4	128	31	20%	23%	53%
LC.big	Lucene	4	4096	31	10%	40%	55%
ER.fst	EasyRec	2	2	7	75%	15%	89%
OE.jep	OpenEphyra	4	4	8	5%	20%	56%

Table 6.1: The OLDI workloads used to evaluate Ubora supported diverse data sizes and processing demands.

6.5.1 Metrics of Merit

Ubora speeds up mature query executions needed to compute answer quality. The research challenge is to complete mature query executions while processing other queries online at the same time. The primary metric used to evaluate Ubora's performance (*throughput*) is mature executions completed per 100 online executions.

Mature executions use resources otherwise reserved for online query executions, slowing down response times. Online queries that Ubora does not select for mature execution (i.e., unsampled queries) are slowed down by queuing delays. We report *slowdown* as the relative increase in response time. In addition to queuing delay, online queries sampled for mature execution are also slowed down by context tracking and memoization.

Finally, we used *true positive rate*, i.e., the percentage of mature results represented in online results, to compute answer quality.

6.5.2 Competing Designs and Implementations

Ubora achieves several axiomatic design goals. Specifically, it (1) speeds up mature executions via memoization, (2) uses a systems approach that works for a wide range of OLDI services, (3) supports adjustable query sampling rate and (4) implements optimizations that reduce network bandwidth usage. Collectively, these goals make Ubora usable. Our evaluation compares competing designs that sacrifice one or more of these goals. They are listed below with an associated codename that will be used to reference them in the rest of the chapter.

- *Ubora* implements our full design and implementation. The sampling rate is set to maximize mature query executions per online query.
- *Ubora-LowSamples* implements our full design and implementation, but lowers the sampling rate to reduce slowdown.
- *Ubora-NoOpt* disables Ubora's optimizations. Specifically, this implementation disables node-local timeouts that reduce network bandwidth usage.
- *Query tagging and caching* essentially implements Ubora at the application level. Here, we implement context tracking by changing the OLDI service's source code so that each query accepts a timeout parameter. Further, we set up a query cache to reuse computation from online execution. This approach is efficient but requires invasive changes.
- *Query tagging* implements context tracking at the application level but disables memoization.
- *Timeout toggling* eschews both context tracking and memoization. This implementation increases each component's global timeout settings by 4X for mature executions. All

concurrent query executions also have increased timeout settings. This is non-invasive because most OLDI components support configurable timeout policies. However, extending timeouts for all queries is costly.

6.5.3 OLDI Services

Table 6.1 describes each OLDI service used in our evaluation. In the rest of this chapter, we will refer to these services using their codename. The setup shown in Figure 6.2 depicts LC.big, a 31 node cluster that supports 16 GB DRAM cache per TB stored on disk. Each component runs on a dedicated node comparable to an EC2 medium instance, providing access to an Intel Xeon E5-2670 VCPU, 4GB DRAM, 30 GB local storage and (up to) 2 TB block storage.

- *YN.bdb* uses Hadoop/Yarn for sentiment analysis. Specifically, it runs query 18 in Big-Bench, a data analytics benchmark [45]. Each query spawns two parallel executions. The first sub-execution extracts sentiments from customer reviews over 2 months. The second covers 9 months. The 9-month execution returns the correct answer, but the 1-month answer is used after a 3-minute timeout. Each sub-execution flushes prior cached data in HDFS, restores a directory structure and compresses its output. As a result, query execution takes minutes, even though customer reviews are smaller than 1 GB. The average response time without timeout is 3 minutes. 44% of queries get the 9-month answer within timeout. We mainly include YN.bdb to show that Ubora can capture answer quality for longer running services too.
- *LC.big*, *LC.wik* and *LC.news* use Apache Lucene for bag-of-words search. All of these workloads replay popular query traces taken from Google Trends.

LC.news hosts 4GB of news articles and books on a Redis cluster with 4GB DRAM. LC.news implements one of the four parallel paths shown in Figure 6.2. It returns the best answer produced within 1 second. Without timeouts, the average response time is 1.22 seconds. Over 83% of LC.news queries complete within the timeout.

LC.wik hosts 128GB of data from Wikipedia, New York Times and TREC NLP [150]. After executing warm-up queries, the data mostly fits in memory. We set timeout at 3 seconds. Without the timeout, response time was 8.9 seconds. 39% of the LC.wik queries complete within the timeout.

LC.big hosts 4TB. Most queries access disk. Average response time without timeout is 23 seconds. The timeout is 5 seconds.

- *ER.fst* uses the EasyRec platform to recommend Netflix movies. It compiles two recommendation databases from Netflix movie ratings [110]: A 256MB version and a 2GB version. Each query provides a set of movie IDs that seed the recommendation engine. The engine with more ratings normally takes longer to respond but provides better recommendations. Query execution times out after 500 milliseconds. 80% of query executions produce the 2GB answer.

- *OE.jep* uses OpenEphyra, a question answering framework [130]. OpenEphyra uses bag-of-words search to extract sentences in the AQUAINT-2 NLP dataset related to queries from the TREC trace [150]. It then compares each sentence to a large catalog of noun-verb templates, looking for specific answers. The workload is computationally intensive. The average response time in our setup was 23 seconds. Motivated by the responses times for IBM Watson, we set a timeout of 3 seconds [34]. Fewer than 15% of queries completed within timeout.

We set up a workload generator that replayed trace workloads at a set arrival rate for each workload. The goal was to keep a concurrency level of 1 at the top-level node. Based on averaging the CPU utilization for all of the nodes used in a workload, our workload generator kept CPU utilization between 15–35% overall.

Table 6.1 also displays numerical characteristics that illustrate the diversity of our tested workloads, including utilization, the quartile coefficient of distribution, and maturity. The utilization shown for each workload is defined as $\frac{ArrivalRate}{ProcessingRate}$. The arrival rate and processing rate used in this calculation were measured for each workload without turning on Ubora. As utilization increases, Ubora is challenged to achieve memoization and replay without creating too much queuing delay. Table 6.1 also shows the quartile coefficient of distribution for the response times of target components in each workload. Finally, we define *maturity* as the ratio between average online query execution time (affected by premature timeouts) and mature execution time. Greater maturity allows less time for mature executions to differ from online executions. These values are computed offline and are used here to characterize the workload. Our services have diverse CPU% and IO% (not shown). This stems from the wide range of data and cluster sizes covered. Taken together, our services represent many OLDI workloads.

6.5.4 Results

Microbenchmark Tests: Our first test studied the effect of data skew and component selection. For this, we used a microbenchmark consisting of three software components, a front component which accepts queries, and two auxiliary components. Each query randomly selected 1 auxiliary component to have a running time X% longer than the other. Here, X%

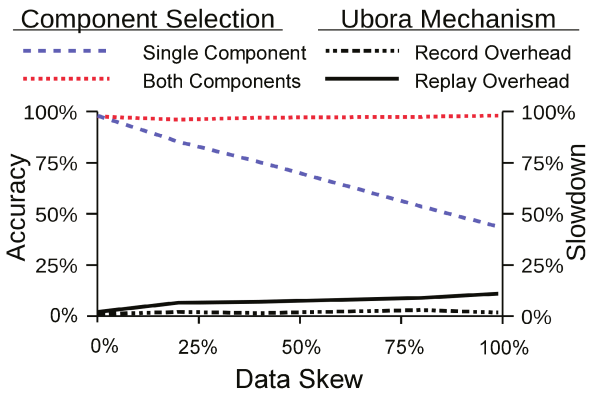

Figure 6.7: Microbenchmark study on the effects of component selection on accuracy and Ubora mechanisms on overhead under changing data skew. Data skew represents the difference in running times between two auxiliary components.

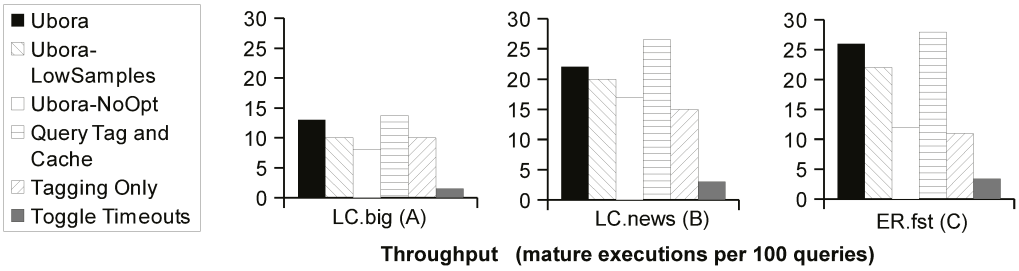

Figure 6.8: **Experimental results:** Ubora achieves greater throughput than competing systems-level approaches. It performs nearly as well as invasive application-level approaches (within 16%).

approximates data skew. The output of each auxiliary component is its running time, and the microbenchmark's output is the largest observed running time. The front component

times out after the shortest component completes (100 milliseconds). We issued 10,000 queries to this micro-benchmark one after another (e.g., 10 queries/second).

The left axis of Figure 6.7 shows the accuracy of mature results in this test, i.e., the relative error between the time given by our mature results, and the running time of the slowest component. We report average error. The top dotted line, labeled *Both Components*, shows results when both auxiliary components are targets. The dashed line shows results when only one auxiliary component is a target. When both components are targets, accuracy ranges between 96-99%. However, the *Single Component* line warns about the perils of selecting targets poorly. Consider the extreme case where the shortest component runs for 100 milliseconds and the longest runs for 200 milliseconds. If the wrong component (in this case, the shorter-running component) was selected, the best possible accuracy is 50%. Record and replay overheads cause further degradation. On the right axis of Figure 6.7, we report slowdown, i.e., increase in response time, caused by record mode and replay mode respectively in the *Both Components* experiment. Record mode includes the cost of redirecting network messages to cache. Its overhead is around 1%. Replay mode includes the cost of extending timeout for the long running component and the cost of queuing delays to replay executions. The slowdown grew by 1.8% per 10% increase in data skew. Effectively, this means that we reduced the amount of time needed to perform a mature execution by 5.5X. These tests show that our record and replay mechanism are implemented efficiently.

Comparison to Competing Approaches: Figure 6.8 compares competing approaches in terms of mature executions completed per 100 online queries. For these experiments, we set the sampling rate to record approximately 40 queries out of every 100. Ubora-NoOpt

reveals that node-local timeouts and just-in-time query propagation collectively reduce the overhead of sampling, improving the throughput of mature execution completions by 1.6X, 1.3X and 2.1X respectively. ER.fst has relatively fast response times which require messages to turn off record and replay modes. Node-local timeouts reduce these costs. Internal component communications in LC.big and LC.wik also benefit from node-local timeouts.

Excluding Ubora, the other competing approach that can be implemented for a wide range of services is toggling timeouts. Unfortunately, this approach performs poorly, lowering throughput by 7-8X. To explain this result, we use a concrete example of a search for "Mandy Moore" in LC.big. First, we confirm that both Ubora and toggling timeouts produce the same results. They produce the same top-5 results and 90% of the top-20 results overlap. Under 5-second timeout, the query times out prematurely, outputting only 60% of top-20 results. Ubora completes mature executions faster because it maintains execution context. This allows concurrent queries to use different timeout settings. Queries operating under normal timeouts free resources for the mature execution. Further, per-component processing times vary within mature executions (recall, Figure 6.3). By maintaining execution context, Ubora avoids overusing system resources. For the "Mandy Moore" query, Ubora's mature execution took 21 seconds in record mode and 4 seconds in replay mode. Conversely, under the toggle timeouts approach, service times for all concurrent queries increased by 4X, exceeding system capacity and taking 589 seconds.

We also compared our systems-level implementation of Ubora, which strives to transparently support a wide range of services, to application-level approaches. For these experiments, we maximize throughput for Ubora on ER.fst. Based on this curve in Figure 6.10(a), we set sampling rate to 20% for all approaches. Application-level approaches can track

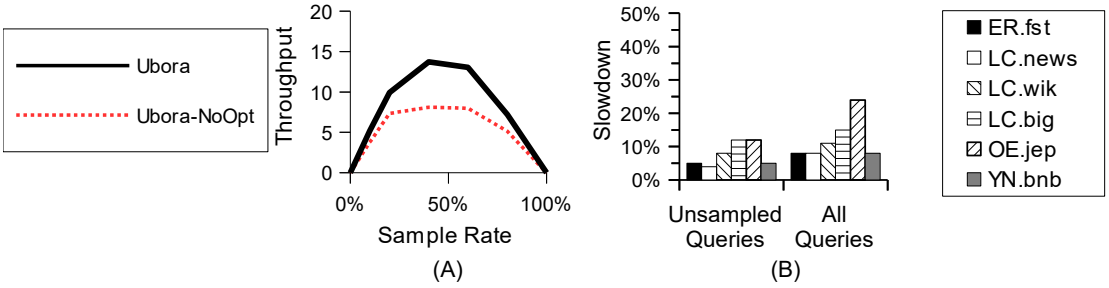

Figure 6.9: **Impact on response time:** (a) Throughput under varying sampling rate for Ubora and Ubora-NoOpt. (b) Ubora delayed unsampled queries by 7% on average. Sampled queries were slowed by 10% on average.

query context efficiently by tagging queries as they traverse the service [35]. Specifically, we modified LC.big, LC.news and ER.fst to pass query context on each component interaction. Further, we implemented a query cache for targeted query interactions [6, 51, 117]. Our cache uses the Ubora's mechanism for memoization but tailors it to specific inter-component interactions and context ids. As such, our application-level approach is expected to outperform Ubora, and it does. However, Ubora is competitive, achieving performance within 16% on all applications. We also compared to a simple application-level approach that disables query caching. This approach shows that memoization improves throughput by 1.3X on LC.big, 1.7X on LC.news and 2.5X on ER.fst. The benefit provided by memoization is correlated with the ratio of mature execution times to online execution times. In ER.fst, mature executions are mostly repeating online executions.

Impact on Response Time: Ubora allows system managers to control the query sampling rate. Figure 6.9(a) compares the throughput rate (mature executions per 100 queries) for

Ubora with and without optimizations, for different sampling rates. Our optimations improve throughput for LC.big by 2X at the 40% sampling rate.

In contrast, Ubora-LowSamples only replays queries when the interarrival time is high. This slight reduction in the sampling rate can still achieve high throughput. However, this approach significantly reduces Ubora's effect on response time. Figure 6.9(b) shows the slowdown caused by the Ubora-LowSamples approach across all tested workloads. By executing mature executions in the background and staying within processing capacity, we achieve slowdown below 13% on all workloads for unsampled queries and below 10% on 4 of 6 workloads for sampled queries. OpenEphyra and LC.big incur the largest overhead because just-in-time context interposes on many inter-component interactions due to cluster size. For such workloads, OS-level context tracking would improve response time for sampled queries.

Collectively, the 5 workloads shown use 9 platforms including widely used Apache Lucene, EasyRec Recommendation Engine, OpenEphyra and NanoWeb PHP server. In general, the variance of mature execution times (i.e., QCoD) correlates positively to the throughput achieved by each workload. The target components in EasyRec workloads in particular have the greatest QCoD. EasyRec workloads yield throughput about 50% relative to other workloads. Higher utilization levels were associated with greater slowdown on unsampled queries, reflecting queuing delay. We also observed less slowdown on ER.fst. This workload had higher maturity and low utilization which limits the potential for slowdown.

Impact of Profiling: Figure 6.10(a)-(b) study our approach to determine sampling rate and front-end components (i.e., memoization). We studied the ER.fst workload. Figure 6.10(a) shows the achieved throughput (mature executions per 100 queries) as the percentage of

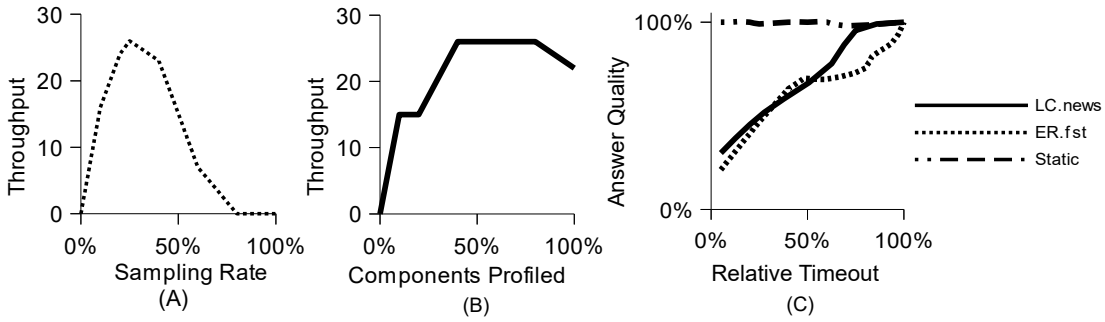

Figure 6.10: **Experimental results for maximized throughput with ER.fst:** (a) We profiled sampling options. (b) We profiled memoization options. (c) Timeout settings have complex, application-specific affects on answer quality.

mature executions initiated increases. Figure 6.10(b) shows the achieved throughput as the percentage of components included as front-end of middle components increases. For the ER.fst workload it is better to apply memoization to many components. The 20% sampling rate for Ubora-LowSamples on ER.fst maximized throughput.

The peak sampling rate corresponds to 12 queries per minute. Because the requests for LC.news took longer, the peak sampling rate for Figure 6.9(a) corresponded to 1 query per minute. First, we observe that under Ubora-LowSamples, the failure rate increases with the sampling rate. This is due to expired cache entries and the potential for additional time between memoization and replay. Under 20% sampling rate, 17% of mature execution fail to yield mature results. This rises to 84% at 80% sampling rate. Figure 6.9(a) agrees with this rise, with 30% of mature executions failing to yield mature results at 60% of the sampling rate. Peak throughput is achieved at the cost of efficiency. We also observe that Ubora's optimizations collectively lead to significant throughput gains across sampling rates.

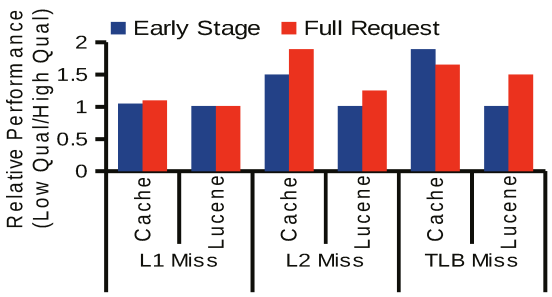

Figure 6.11: Some hardware counters predict answer quality.

Studying Answer Quality: Figure 6.10(c) shows answer quality (i.e., the true positive rate) as we increase timeout settings. For LC.news and ER.fst, we increase timeouts at front-end components. We also validate our results by increasing timeouts in an unrelated component in ER.fst (Static). We observe that answer quality is stable in the static setting. Further, answer quality curves differ between applications. After timeout settings reach 600 milliseconds for LC.news and 300 milliseconds for ER.fst, the curves diverge and answer quality increases slowly for ER.fst. Finally, answer quality curves have 2 phases in LC.news and 3 phases in ER.fst. It is challenging to use timeouts to predict answer quality.

Using Hardware Counters to Improve Sampling: Online executions that complete without triggering timeouts make mature executions unnecessary. Ubora may further reduce its overhead by turning off memoization and replay when it predicts online executions will complete fully. Prior work has shown that hardware counters are useful predictors of query outcomes. We studied whether hardware counters collected early in a query execution can be used to predict answer quality in LC.big. For this test, we used the Google trace and issued queries one at a time under a tighter timeout (3 seconds). We collected level-1 cache

misses (L1), level-2 cache misses (L2), and translation lookaside buffer (TLB) misses every second. Figure 6.11 shows hardware counters after the first 1/3rd of query execution and across the whole query execution. The figure shows the results of low quality queries relative to the results for high quality queries. We observed the ratio of L2 misses and TLB misses on cache nodes were markedly higher in cache nodes. These predictors detect high-quality queries quickly enough to prevent mature executions (if the query had been sampled). In LC.big, this approach has the capacity to reduce mature executions by a further 60%, doubling Ubora's throughput.

For our Apache Lucene search engine, this makes sense as TLB misses often mean that slow Lucene disk will have more lookups and that the query will likely timeout.

6.6 Online Management

OLDI services use anytime algorithms, returning valid results even when provisioned to provide target response times. In addition to classic metrics like response time, these services could use answer quality to manage resources. We show here that Ubora enables better resource management through answer quality. In this case study, we use Ubora to improve admission control, a classic system management challenge.

Control Theory with Answer Quality: We studied admission control on the LC.big workload. We issued two classes of queries which arrived at different TCP ports, indicating high and low priority. High priority requests arrived at a fixed rate in terms of requests per second. We used diurnal traces from previous studies [140, 144] to issue low priority requests. At the peak workload, low and high priority arrival rates saturate system resources (i.e.,

utilization is 90%). Figure 6.12(a) shows the *Arrival Rate* of low priority queries over time as well as the number of low priority search requests admitted. We used Ubora to track answer quality for high priority queries. Here, answer quality is the true positive rate for the top 20 results. At the 45 minute and 2 hour mark, the query mix shifts toward multiple word queries that take longer to process fully. This acccounts for the drops in answer quality for the *No Sharing* line in Figure 6.12(b). When quality dipped, we decreased the admission control rate on low priority queries. Specifically, we used a proportional-integral-derivative (PID) controller. Every 100 requests, we computed answer quality from 20 sampled queries (20% sample rate). The PID controller used 10-minute sliding windows to average out spikes in answer quality and timeout frequency. The PID controller weighted current reading at 40% (i.e., proportional portion).

The y axis of Figure 6.12(b) shows answer quality of competing admission control approaches. When no low priority queries are admitted, the *No Sharing* approach maintains answer quality above 90% throughout the trace, even during periods with complex queries. When admission control is disabled, the *Full Sharing* approach sees answer quality as low as 20%, corresponding with peak arrival rates. The PID controller powered by Ubora manages the admission rate well, keeping answer quality above 90% in over 90% of the trace. There is about a 20% drop in answer quality for the UBORA PID controller approximately at the point in time where the query mix increases in complexity. The drop in UBORA TPUT occurs concurrently with this, indicating that the amount of low priority queries shed to counter this drop increased. It allows almost 60% of low priority queries to complete (*Ubora (TPUT)*).

The state of the art for online management in OLDI services is to use proxies for the answer quality metric. Metrics like the frequency of timeouts provide a rough indication of answer quality and are easier to compute online [66]. For comparison, we implemented a PID controller that used frequency of timeouts instead of answer quality. We tuned the controller to achieve answer quality similar to the controller based on answer quality. However, timeout frequency is a conservative indicator of answer quality for Lucene workloads. It assumes that partial results caused by timeouts are dissimilar to mature results. Figure 6.12(a) also shows that the controller based on timeout frequency (*TO Freq (TPUT)*) drops requests too aggressively. Queries can only be dropped explicitly in our system, so both TO Frequency and Ubora PID controllers achieve full throughput on high priority requests. For most of the trace, the Ubora PID controller has a higher throughput on low priority requests than *TO Freq*. When arrival rate increases both Ubora and TO Frequency controllers admit fewer low priority queries. The *TO Freq* PID controller is consistently more conservative than *Ubora PID*. The TO Frequency PID controller only allowed 25% of low priority queries to complete. Compared to the TO Frequency PID controller's peak throughput over the whole trace, our Ubora PID controller improved peak throughput on low priority queries by 55%.

Sampling Rate and Representativeness: Ubora allows reducing the overhead of mature executions by sampling online executions. This lowers mature results per query, but how many mature results are needed for online management? Table 6.2 shows the effect of lower sampling rates on the accuracy of answer quality measurements and on the outcome of adaptive admission control. We observed that sampling 5% of online queries significantly increased outlier errors on answer quality, but our adaptive admission control remained effective—it still achieved over 90% quality over 90% of the trace. In contrast, a 2% sampling rate produced many quality violations.

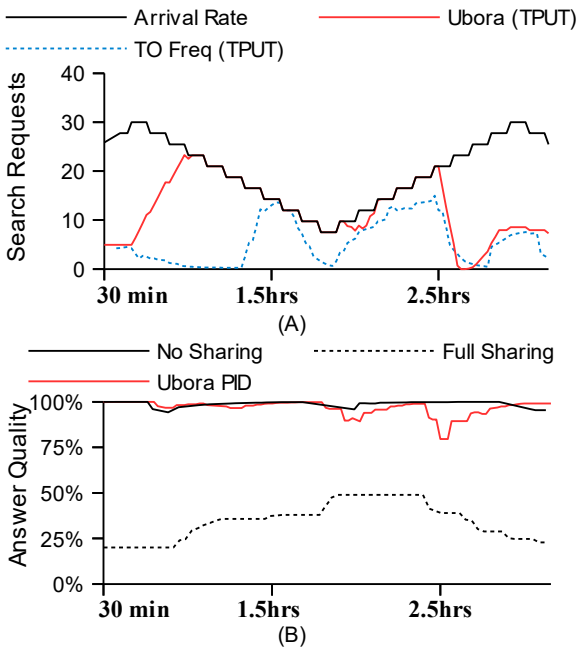

Figure 6.12: Ubora enables online admission control. Arrival rate refers only to low priority requests. High priority requests arrive at a fixed rate.

6.7 Related Work

Ubora focuses on online systems, which trade answer quality for fast response times. Zilberstein first characterized similar applications as anytime algorithms [177]. Like the online, data-intensive workloads used with Ubora, anytime algorithms increase in result quality as they increase in computation time. The metric Zilberstein uses closest to our answer quality metric is accuracy, but does not indicate how the exact answer is to be reached for comparison. His work indicates that anytime algorithms should have measurable quality, monotonically increase in quality as computation time increases, provide diminishing returns, and produce a correct answer when interrupted. Ubora broadens the category of

Sampling Rate	Measurement Error for Answer Quality		Rate of Quality Violations
	Avg.	95th %tile	
10%	0%	0%	4%
5%	20%	45%	9%
3%	30%	50%	13%
2%	51%	78%	29%

Table 6.2: Adaptive management degrades under low sampling rates. A *quality violation* is a window where answer quality falls below 90%. Error is relative to the 10% rate.

applications that can use an answer quality metric beyond anytime algorithms, not requiring that applications can suspend and resume at any time, nor requiring that the optimal answer be determined in constant time.

6.7.1 Approximation for Performance

Recent work has focused on introducing approximation into existing systems in order to increase performance [46, 67, 66].

ApproxHadoop [46] integrates sampling of input data, user-defined approximate code versions, and execution of a subset of tasks into Hadoop. ApproxHadoop allows the user to set error bounds within a confidence interval, set a specific data sampling ratio, or specify the percentage of tasks to drop in order to increase performance. Ubora enables users to similarly manage resources based on the online answer quality trace.

Sequential search may terminate early on a server if the processing of the ranked documents goes below a certain relevance. Since parallel search over the same index will generally result in more processing per query, [67] reduces this wasted work by keeping the order in which documents are processed sequential. While this is not necessary under low load, higher loads are more impacted by wasted work. The authors adaptively change the amount of parallelism per query based on the current system load.

Kwiken is an optimization framework for lowering tail latency in Bing [66]. Kwiken uses techniques which include allowing return of incomplete results, reissuing queries that lag on different servers, and increasing apportioned resources. It calculates incompleteness as utility loss based on whether the answer returned contains the highest ranked document for certain stages, and in other stages this is the percentage of parallel components that had not responded. Our work differs from their solution in that we focus on speeding up the mature execution with which to produce answer quality. Additionally, our framework provides for provisioning of other resources based on answer quality.

In between executing queries, DICE uses wait time to speculatively execute the queries most likely to be asked next, and cache these results [68]. DICE also implements time-outs on total query execution, so that even if only some of the data is assembled in post processing, an answer will be available. DICE sampled data proportionately to the most likely speculative queries found. DICE is very similar in two ways to Ubora, in that we also use cached data from queries hidden from the user. However, DICE uses this cached data to improve the latencies of further queries within a user session rather than for mature executions. DICE also uses sampling to reduce the resources spent running queries not sent by an end user.

6.7.2 Query Tagging

One of the approaches we used to increase the maturity of answers as an alternative to Ubora is specially tagging each query with context clues. Several recent works have illustrated the use of query tagging for approximated workloads.

A proxy-based approach can dynamically scale quality of web results across different end platforms [38]. [38] uses lossy compression to distill specifically typed information down to the portions with semantic value. Their proxy adapts on demand to fit the needs of a client. [38] accesses a mature execution from a web server and approximates this data to meet the needs of a range of client platforms. We instead focus on services that provide online results, and measure the amount of approximation present.

SocialTrove tags queries with data regarding the minimum diversity expected among the returned samples from data-intensive applications. Instead of measuring answer quality, SocialTrove uses application-specific similarity metrics to automatically cluster and summarize social media data [5].

[56] uses a budget consisting of total execution time for current queries to determine whether and how long to schedule a query. Each query is tagged with an amount of processing time based on this budget. Their work uses a feedback mechanism to help ensure the desired response times are being met, and an optimization procedure to schedule based on request service demands and response quality profiles. Their algorithm takes advantage of prior knowledge regarding the overall concave quality profile of Microsoft Bing to estimate the individual request quality profile, rather than attempting to measure request quality with a mature execution.

Similarly, Zeta was designed to better schedule requests in online servers for high response quality and low response quality variance [54]. Zeta focuses on online services that produce partial results under a deadline, where trading additional computation time produces diminishing returns in additional response quality. Their response quality, like our answer quality, uses an application-specific metric to compare a partially executed request to a full execution. They measure their response quality offline.

[123] tags queries with deadline and arrival time to implement their Fast Old and First (FOF) algorithm, which schedules incoming, unknown requests on the fastest core available in a heterogeneous processor, then migrates requests from slower cores to faster cores as jobs finish. They explain how heterogeneous processors can execute long requests on faster cores and shorter requests on slow cores to achieve high throughput and high quality. Their algorithm can improve answer quality and throughput in heterogeneous processors as compared to homogeneous processors with the same power budget. The authors used Bing without deadlines in a controlled setting to produce mature executions, and then used this data in their simulation study.

6.7.3 Timeout Toggling: Adaptive Configuration

A second approach that can be used to achieve mature executions in an online setting is to dynamically change the configuration at the application level, via argument specification. Earlier in this chapter, we compare Ubora to timeout toggling, which uses this approach to extend query processing time.

As in our work, [75] focuses on online data-intensive computations occurring across multiple components working together. Their system changes configuration to maximize a utility function, then displays interpreted system responses to the user and corrects computations when the user indicates incorrect analysis.

Our work focuses on data-intensive applications, and uses application-specific similarity metrics to study answer quality. Previous work has used answer quality to reduce costs in cache provisioning for online, natural language applications [70]. However, their work adapted cache configuration offline based on answer quality.

ISPEED uses a deadline-agnostic scheduler to explore anytime algorithm workloads without information regarding individual queries [176]. ISPEED focuses on maximizing total utility over all jobs in the cluster, and ignores concerns regarding individual query deadlines. In addition to the utility functions used in [56, 54], ISPEED also facilitated a user study for the Google search engine to find its average utility function [176].

SkewTune mitigates skew for user-defined MapReduce programs by reconfiguring the amount of data per task online [81]. SkewTune has similar goals to Ubora with regard to transparency and minimal overhead for untuned queries, but dynamically redistributes data from the task expected to take the longest to complete instead of using approximation. Our work shows that data skew is one of the motivations for approximation in online, data-intensive services.

6.7.4 Adaptive Resource Allocation

Also highly related to Ubora is the area of adaptive resource allocation [139, 41, 83]. [139] presented a library for estimating resource demands with seven different approaches. Using their library, it is possible to control how often the resource use is sampled, when and for how long to perform the estimate.

DC2, an autoscaling cloud service, can learn an application's system parameters and scale based on its understanding of resource requirements [41]. Without direct knowledge of the application's needs, DC2 relies on user-specified SLA information, virtual cpu statistics, and knowledge of request URLs to autoscale. Like Ubora, DC2 is mostly transparent, with key information provided by the user. However, DC2 focuses directly on autoscaling.

AROMA is an automated resource provisioning system which uses Hadoop parameter configuration and resource allocation in a hetergeneous cloud environment to target quality of service while minimizing cost [83]. Instead of directly profiling each workload and regulating resources based on answer quality, AROMA profiles each workload for a short time on a staging cluster before matching the workload's signature to a cluster of workloads with a set of associated resources.

Chapter 7: Rapid In-situ Characterization for Co-Located Workloads

Workloads save on operating costs by using colocation services. Google Cloud Services, Microsoft Azure, and Amazon EC2 offer competitive opportunities for colocation, with varying resources [4, 50, 113]. While virtualized cores are shared between colocated workloads, each receives a set amount of memory according to the type of instance requested. Because of the colocated nature of these workloads, cloud services offer these opportunities at lower cost than their fixed performance instances, as shown in Figure 1.1.

The goal of colocation for cloud providers is to increase physical machine utilization while minimizing impact of resource sharing on workload performance [135]. Service level agreements expect that during more than 99% of a workload's lifetime, it will behave within specific parameters, e.g., a response time bound. Despite best efforts, this cannot be 100% for colocated workloads because these workloads share resources to keep costs low. With so many opportunities for sharing resources available, characterization is important to choose the potential placements least likely to lead to Service Level Objective (SLO) violations. Characterizations trade time collecting traces for accuracy, but the effects of this are not usually studied. Most schedulers (e.g., Paragon, Agile) collect traces for a given time period before making a scheduling decision [24, 111].

The list of features collected in these traces is also usually static. Agile, for instance, only collects features easily accessible through the /proc interface [111]. However, modern virtual machine and container software have access to many additional features such as hardware counters through the machine state register (also called model-specific register) interface.

In this chapter, we present Quikolo, our cloud service implementation that enables an already colocated workload to test new placements for better resource sharing (lower SLO violations). Quikolo uses the Kubernetes API to launch workloads to new colocation environments. A system-level component of Quikolo then characterizes the speculative colocation opportunity according to a rich set of process-granularity features combined with architecture-level hardware counters. Quikolo uses this data to make a recommendation to the requesting workload regarding the expected latency on the characterized environment. Kubernetes users can then deploy their container pods to locations expected to grant them high utilization with manageable resource contention.

We designed Quikolo to allow us to study the effects of dimensionality and collection time on characterization accuracy in colocation environments. We use ensemble modeling to explore the effects of feature dimensionality on accuracy. Each dimension of the feature vector is used as a model input to our ensemble, allowing us to study the effects of including each feature in our characterization. Quikolo uses high dimensional feature vectors to tune which models are most expressive of the current workload. By using ensemble modeling, Quikolo can reduce the number of features collected by 78% for repeat customers.

Incoming request workloads set a budget in terms of time regarding feature collection time. We can reduce collection time using statistical convergence. In other words, we stop collecting characterization data when 80% of feature aspects have reached statistical convergence. Statistical convergence allows us to study the effects of collection time on characterization accuracy by deterministically scaling how close to the standard deviation from previous readings we want the latest feature vector to fall. By using statistical convergence, we can reduce collection time by 20%.

This chapter presents the following contributions:

1. We developed Quikolo, a cloud service which enables speculative deployment of requested workloads using the Kubernetes API. Quikolo then characterizes the colocation environment in terms of expected latency using process-granularity statistics and hardware counter readings. Kubernetes users can use the expected latency in the new colocation environment to deploy container pods to desirable locations.
2. We use ensemble modeling to study how increased dimensionality improves characterization accuracy within colocated environments, even at reduced collection time.
3. We use statistical convergence to study how accuracy improves with increased collection time, even with low dimensionality.

The chapter is laid out in the following sections. Section 7.1 details how Quikolo works, with subsections on collecting features and Service Level Objective data, as well as our test workloads. Section 7.4 presents our study of collection duration using statistical convergence. Section 7.5 presents our study of feature dimensionality using ensemble modeling. Section 7.6 explains how this work relates to papers in the systems community.

7.1 Design

We designed a solution to characterize in-situ in colocation environments. To do this effectively, this solution had to have low overhead. To be useful, it had to be low cost to use. We also instrumented our solution to allow study of characterization duration and feature selection. Once a workload is deployed, we continuously acquire instantaneous program counter readings which are analyzed for expected latency. We collect k features for a minimum of 1 minute and at most T minutes, but may stop early if statistical convergence indicates diminishing returns on accuracy increase. We use ensemble modeling to determine which features contribute most to latency prediction accuracy.

7.1.1 Design Goals

- Low Client Cost: Workloads using our service must set a budget in terms of the maximum minutes to collect features. A decision on speculative latency must be reached before this maximum characterization duration.
- Low Colocation Overhead: Colocated workloads increase resource contention. We want to minimize the impact of characterization on this resource contention. To this end, while collection is done in-situ, all data is stored and processed on a different machine. We further lower overhead by reducing duration and features needed to obtain accuracy.
- Reduce Duration: We study the effects of characterization duration on decision accuracy. In order to do this, we terminate the speculative workload when feature distributions statistically converge.

- Choose Features Wisely: We study the effects of increased features on decision accuracy. To do this, we introduce a mask variable which controls the features collected. Ensemble modeling identifies the features most useful in determining speculative latency. After a workload has been seen and characterized once, we store the information regarding which features were most useful in this mask variable. In future uses of this service, only those useful features are collected.

7.1.2 Design Parameters

The design for in-situ characterization of speculative deployments in colocation environments has three components: the client, the controller, the collector, and the analyzer.

- The Client is used to inform the controller of existing colocation deployments that would like to be characterized in a different location. The controller accepts a budget b in terms of minutes, the IP address and port of the currently running workload, and a description of the workload sufficient for deployment.
- The Controller accepts requests for speculative deployment, tracks speculative deployments currently in the list of available colocation environments, and randomly assigns the requesting workload a new colocation location. The information regarding this speculative deployment is sent to two collectors, the collector on the colocation environment where the workload is currently running and the collector on the colocation environment where the workload was speculatively deployed.
- The Collector receives two types of requests, SLO data collection and resource data collection. In order to collect SLO data, the collector needs the IP address and port where the

original workload runs, as well as the IP address and port where the speculative deployment expects requests. When an SLO data collection request is received, the collector performs the job of duplicating and forwarding copies of all queries received by the original workload to the speculative deployment. The collector also logs arrival times and response times, to measure latency. The second type of request, resource data collection, does not require information regarding IP addresses and ports, but does require data regarding the process IDs assigned to the speculative deployment. The collector can then use the available tools to characterize the process IDs relating to this request. To test the effects of increased features on decision accuracy, the collector also receives information regarding which features to test for this speculative deployment. For each request type, the collection ends when the budget for that request is entirely used, or a message from the Analyzer indicating statistical convergence has been received.

- The Analyzer receives continuous updates from the SLO data collection and resource data collection requests. These streams of timestamped data are correlated and checked for statistical convergence. When the budget has been entirely used or statistical convergence has occurred, the analyzer then uses machine learning to build a suite of models with different feature sets, and uses the best performing models (tested on a subset of the collected data) to project latency for extrapolated feature data. This process is well described in [111] and is not the focus of our study. Instead, we use the machine learning models to identify and record which features were most relevant to the final assessment regarding whether the speculative deployment resulted in lower latency than the original workload location.

7.1.3 Design Limitations

Speculative deployment and in-situ characterization of colocation environments works in colocation environments which do not show preference for newly arriving deployments. In environments such as Amazon EC2, which allocates newly deployed workloads more credits (i.e., resources), the speculative deployment will not be an accurate approximation of the colocation environment over time. In order to work for this environment, the minimum budget assigned must be sufficient to exhaust the extra credits before accurate characterization can proceed.

Additionally, the mechanism used for identifying SLO violations works only for workloads which receive requests over an IP connection, with request latencies lower than 1 minute. Longer requests require more budget for characterization. Workloads that only receive data or requests and do not return processed results will not be accurately characterized with respect to latency, since the return message is timestamped to calculate this.

7.2 Quikolo Implementation

We chose to implement Quikolo to deploy workloads using Kubernetes with Docker containers, which allows workloads to colocate on the same physical machine with arbitrary runtime environments. We assume that this machine exists as one of a pool of physical machines that run cloud services, and that when spot instances are available on another machine in the pool, a workload already running may choose to bid for this spot instance, hoping that moving may result in less resource contention. Quikolo could be instrumented to launch workloads to other colocation environments.

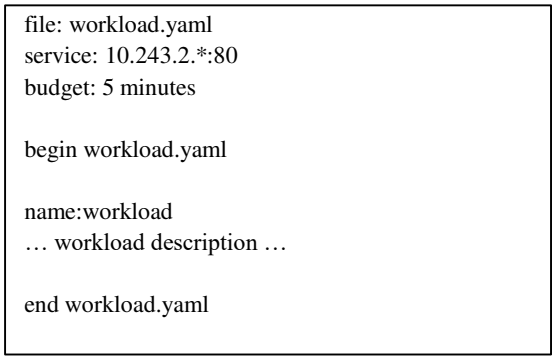

Figure 7.1: Quikolo request

Workloads use a TCP message to issue a request to the Quikolo service. The message contains a YAML file which Kubernetes can launch, the current IP address and port on which the workload accepts requests, and a budget in terms of minutes the workload will pay to be speculatively deployed and characterized. An example request is shown in Figure 7.1. Quikolo uses the Kubernetes API to launch the pod described in the YAML file. Quikolo modifies the pod name, discovered through the YAML file, to query Kubernetes regarding deployment location and process IDs. Quikolo queries MySQL to determine if the unmodified pod name has been characterized previously, and if so, which features to collect. Quikolo's workflow can be found in Figure 7.2.

Quikolo uses nftables to record request arrival times and data packets heading to the current workload. These cached requests are then issued to the speculative deployment of the workload. The process IDs are used to collect and aggregate process-specific data regarding CPU, memory, disk, and network I/O at the system level. Machine-wide statistics are

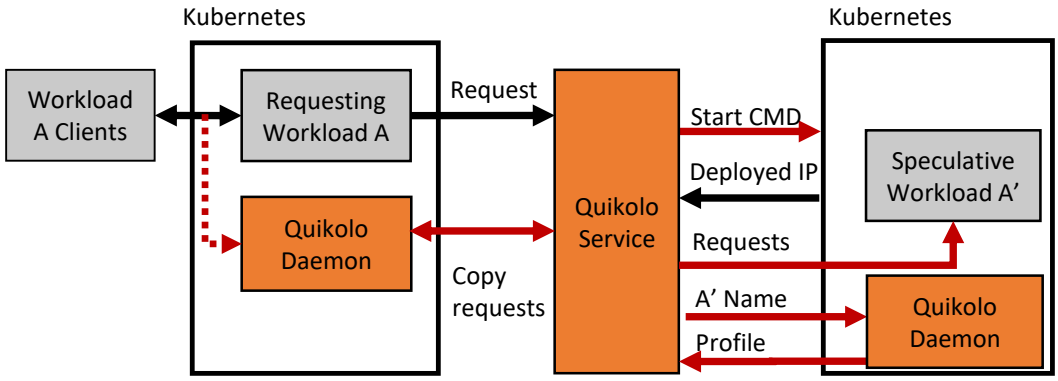

Figure 7.2: Workflows in Quikolo. Arrows reflect messages in execution order (top to bottom). Dotted lines represent messages seen by the speculative workload but not by the original workload or clients.

collected using the /proc interface at the system level and cache information is added using hardware counters at the architecture level.

7.2.1 Feature Collection

In our setup, a Quikolo daemon runs on each machine running Kubernetes. This daemon receives a TCP message when a workload requesting characterization is running on their machine. This TCP message contains the name of the workload, feature mask, budget, and whether the workload on this daemon's machine is the original or speculative.

Arrival and Service Rate

When a Quikolo daemon receives word that the original workload is on their machine, the daemon uses Kubernetes knowledge regarding IP address and port with an nftables command to record incoming request arrival times and data packets. These data packets

and arrival times are cached in MySQL on the Quikolo master node, and issued as requests to the speculative workload. Service times are recorded at this fake client and cached in a MySQL container colocated with Quikolo master pod.

System-level Statistics

When a Quikolo daemon receives a message with the identifying pod name of a speculative workload that is on their machine, the daemon uses Kubernetes knowledge regarding the process IDs to query the /proc interface regarding allowed CPUs, cycles used, memory and disk bytes transferred, and network packets sent or received. To get the process IDs relevant to a specific pod name, we use ”docker ps” to acquire a list of containers currently running on that machine. Container names starting with ”k8s” belong to Kubernetes, and the pod name also be embedded in the container name. These container names can be mapped directly to processes running on the machine, which allows process-specific data collection from the /proc/$PID interface. This process-specific data is aggregated across all relevant processes when features are recorded. Machine-level statistics are also collected using the /proc interface so that a ratio of workload usage compared to total machine resource usage can be represented. All of this feature data is cached in MySQL with a millisecond-granularity timestamp.

Architecture Statistics

The Quikolo daemon on a machine executing a speculative workload also collects data through the Intel PCM library at the Machine State Register level. These hardware counters inform Quikolo regarding per-L2 cache miss rates and per-L3 cache miss rates. This data

is also cached in MySQL, and used in conjunction with the information regarding allowed CPUs to further refine which features are representative of the speculative workload.

7.2.2 Organization

A thread on the Quikolo master is assigned to each request coming from a client workload. This thread is responsible for communicating with the Quikolo daemon on nodes where the original workload and speculative pod are deployed, and also correlates real-time latency and resource usage data streamed from MySQL using their millisecond-granularity timestamps. At each timestep, new data is added to the feature traces, which are then checked for statistical convergence as described in Section 7.4. If statistical convergence is found, termination messages are sent to the Quikolo daemons; otherwise, termination messages are sent to the Quikolo daemons at the end of the budget set by the client. To increase fault-tolerance, the thread on the Quikolo daemons handling this request also set a time out equal to the budget. This ensures that the speculative deployment will never be characterized for more than their budgeted amount.

After the characterization finishes, the timestamp correlated data traces are analyzed by the same thread in the Quikolo master. We build an ensemble of models to predict expected request latency using subsets of features collected, as described in Section 7.5. Using wavelet analysis as described in Agile [111], resource usage data streams are projected into the future. Our ensemble of models is tested on these projected data streams.

7.3 Experimental Evaluation

In this section, we describe the setup for our experiments, including the architecture and workloads. We study the overhead caused by Quikolo in this section before moving on to our study of characterization duration and feature selection.

7.3.1 Architecture

Our cluster has 4 CentOS nodes running Kubernetes and Docker. All of our characterization experiments that rely on using Intel PCM counters are done on Intel Xeon C5-2660 running at maximum 2.2 GHz. Three machines in the cluster are instead Intel Xeon X3330 running at maximum 2.66 GHz, which are not supported by Intel PCM. These machines run the Kubernetes master and the originating Kubernetes pod acting as the Quikolo client. The Quikolo master runs on the same node as the Kubernetes master, and Quikolo daemons run on each node in the cluster.

7.3.2 Workloads

The objective of Quikolo is to explore the accuracy of characterizing rapidly in situ on colocation environments. To test the above designs, we set up batch scientific workloads to run on a 16 core Intel Xeon server, and injected services with varying arrival rates. We use the following interactive workloads to test the above designs:

- *Lucene (LS)* is a search engine service which responds to user queries with subsecond (<3 ms average) response times. We characterize this read only service at low arrival rate (400 queries / second) and at high arrival rate (800 queries / second).

- *Redis (RS)* is a key value store service which responds to user queries with subsecond (<4 ms) response times. We characterize this read/write service at low arrival rate (250 queries / second) and at high arrival rate (500 queries / second).

We use the following background workloads, running continuously, to colocate with our interactive workloads:

- *Conjugate Gradient (CG)* is a batch scientific workload which computes the smallest eigenvalue of a positive definite symmetric matrix. It is known for its irregular memory access patterns, because the matrix in this data set is large and sparsely populated [53]. It exhibits long (>30 second) response times.
- *Embarrassingly Parallel (EP)* is a batch scientific workload which computes embarrassingly parallel tasks such as map reduce jobs. It tests the limits of floating point performance, and has no significant communication between cores. It exhibits medium (between 8 and 20 second) response times.
- *Fourier-Transform (FT)* is a batch scientific workload which solves a 3D partial differential equation. It is representative of spectral computations, with all-to-all communication. It exhibits medium (between 8 and 20 second) response times.
- *Integer Sort (IS)* is a batch scientific workload which computes integer search. It is known for random memory access, and tests the speed of integer computation. It exhibits fast (<5 second) response times.
- *Lower-Upper Gauss-Seidel solver (LU)* is a batch scientific workload which simulates computational fluid dynamics using a system of nonlinear partial differential equations. It contains a limited degree of parallelism compared to the other workloads we examine. It exhibits long (>30 second) response times.

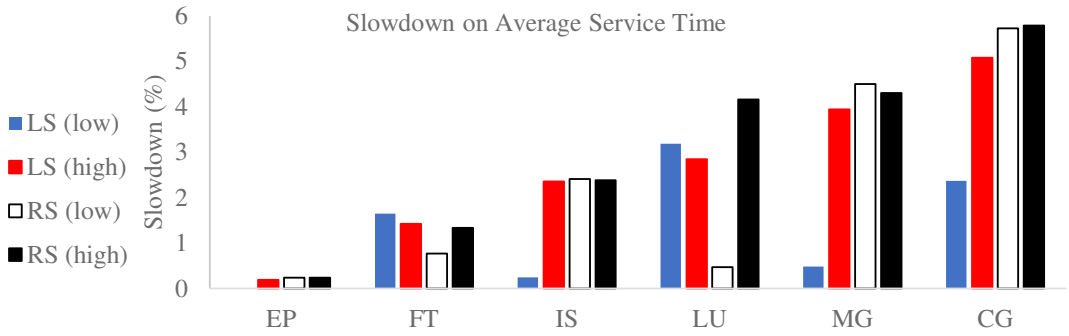

Figure 7.3: We show overhead with and without SLO redirection, for Quikolo using all features. (a) Each workload executes in isolation. (b) Global slowdown for varying colocation mixes.

- *Multi-Grid (MG)* is a batch scientific workload which uses V-cycle multigrid method to solve 3D scalar Poisson equations [11]. It features long and short distance communication. It exhibits fast (<5 second) response times.

7.3.3 Overhead

One of the goals of our design was to keep overhead low. Both the Quikolo daemons cause overhead, first from collecting data on features, and second from additional traffic on the network due to SLO data collection. *Slowdown* (response time using Quikolo vs response time without Quikolo) is measured individually on the original workload being speculatively deployed and globally for workloads on the machine where the workload is speculatively deployed.

We studied the slowdown resulting from collecting all available features on the speculative colocation environment. Lucene at low utilization has a 35% slowdown on average across

all colocation environments. At high utilization, Lucene is less likely to have its resources allocated to other processes during idle cycles, so slowdown was negligible. Slowdown on Redis was also not statistically relevant on any colocation environment. However, the background workloads paid a higher overhead due to Quikolo, as shown in Figure 7.3. This is primarily because the NAS parallel benchmarks are using resources on all available cores, whereas the Docker containers limit the services to a single core by default. Like Redis, the EP (Embarrassingly Parallel) workload has high parallelism with data independence; other workloads tested have more data interdependence, so when a single thread slows down, the entire workload is affected.

In our studies on characterization duration in Section 7.4 and Section 7.5, we introduce mechanisms that reduce overhead, but when this is done it also reduces the accuracy expected from the colocation speculative deployment service.

7.4 Duration Study

Our study of collection time on characterization accuracy required that we allow Quikolo to dynamically scale the amount of time we allow a speculative workload to process. Traditionally, features for characterizations are collected for a fixed amount of time (T). Instead, we allow a speculative workload to end early when we determine that the features for this workload have statistically converged.

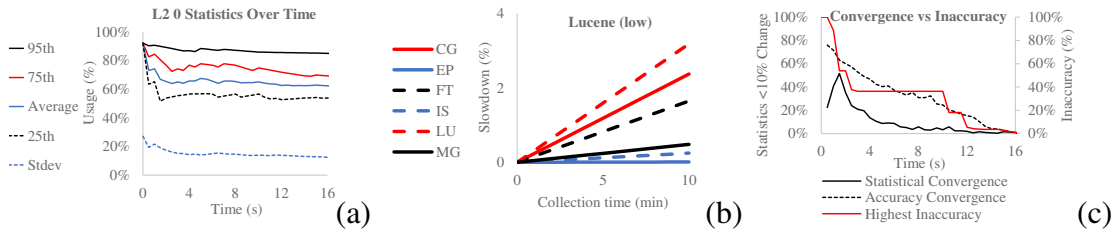

Figure 7.4: Feature change decreases as Lucene characterization progresses. (a) Statistics at time t for L2 cache 0 ($f(16)$). (b) Overhead on the colocation environment during a 10-minute window as collection time increases. (c) Stepwise function RQ shows across all features, the percentage of statistics changing more than 10% compared to the previous feature readings. Accuracy convergence indicates the percentage of statistics greater than 10% change from the final statistics calculated over the entire trace. Highest accuracy shows a comparison of the highest percent difference at each feature reading from the final statistics calculated over the entire trace.

7.4.1 Statistical Convergence

We consider the features for a workload to have statistically converged when the new feature readings consistently fall within expected distributions. At each time t that new readings occur, we calculate statistics such as median, average, standard deviation, and percentiles for each feature. We compare these statistics to prior statistics for this workload that do not include the most recent readings, using percent difference. Other functional methods of determining change in expected distributions exist, and may be swapped out arbitrarily. When any one of these statistics shows a change less than a predefined similarity threshold $s\%$, it has converged. When more than $s\%$ of the statistics for a feature have converged, that feature has converged. We collect statistics for each workload for at least 1 minute, but stop collecting statistics before the budgeted time if $s\%$ of the features have converged. After this point in time, we are no longer attaining feature information that will substantially change a latency decision.

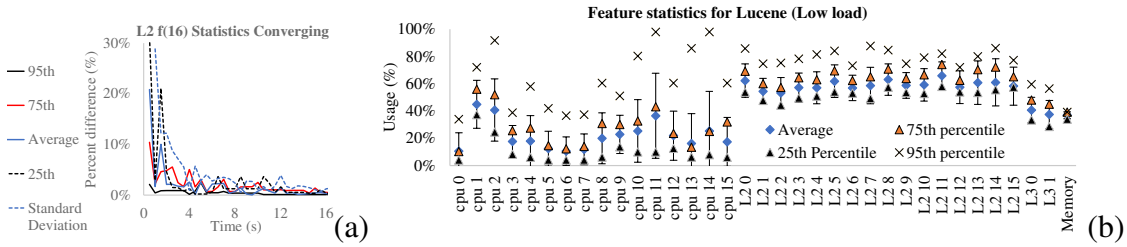

Figure 7.5: **Which Statistics Converge:** (a) Percentiles were less susceptible to outlier readings than standard deviation. (b) Standard deviation (bars in chart) describes features in a way percentiles do not capture.

Algorithm

The inputs to our algorithm include the workload as a Kubernetes YAML file, the expected collection time T, and the similarity threshold $s\%$. The expected collection time should be the number of minutes that the user would normally spend characterizing the workload. The similarity threshold should be a percentage representing how accurate the user would like the output to be, compared to a full run at collection time T.

Characterizations are completed in a single run collecting traces all the available features, including CPU cycles, L2 and L3 cache misses, memory, disk, and I/O usage. Equation 1 shows that our goal is to keep the time t that we spend collecting traces much less than the expected characterization period T. The algorithm determines when to stop collecting features for the input workload.

$$RT(c) = t \ll T \tag{7.1}$$

Currently, we characterize a single speculative deployment per request. It would be possible in the future to maximize the number of speculative deployments launched per request, bounded by the request budget, but that problem is orthogonal to this work.

The output from each speculative deployment c is a set of traces, one trace for each of n available features, as reflected in Equation 2. According to Section 7.5, subsequent speculative deployments of the same workload will only collect a set of traces for the most relevant features.

$$c \longrightarrow f_1, f_2, f_n \tag{7.2}$$

In our algorithm, we use multiple statistics $F(f_n)$ calculated from the instantaneous readings of each feature trace. For instance, Figure 7.4(a) shows continuously updated average and percentiles for the L2 cache belonging to CPU 0 while the Lucene workload described in Section 7.1 runs. These statistics are used to help calculate change between feature distributions with and without the most recent feature readings. Equation 3 explains how features are filtered through these statistics to get change, using percent difference. Other definitions for $I(f, i)$ are valid and may be used so long as change in feature distribution is represented.

$$I(f, i) = \frac{|F(f, i) - F(f, i-1)|}{|F(f, i-1)|} \tag{7.3}$$

We use percent difference to determine feature distribution change between one set of feature readings and the next. We then use a threshold function to determine whether a feature

statistic has reached the requested similarity s. This step function $\Delta(f, i, m)$ returns 1 if statistic m has produced values for feature f_n within $s\%$ of the values produced without the most recent feature reading. Figure 7.4(c) shows percent difference converging for the Lucene workload described in Section 7.1. The spike at 6.5 seconds indicates entry into a phase with high network use, which resulted in widespread change at most levels of the memory hierarchy. Shortly after this phase begins, the feature readings show less than 5% statistical change. To demonstrate the overhead saved by converging early, we simulated early convergence for Lucene at low load on each of the NAS benchmarks running in the background. Figure 7.4(b) shows that slowdown for MG is 48%, slowdown for IS is 25%, and slowdown for EP is 0.7%. This low speedup is because IS and MG are the fastest task types to complete and EP has low data interdependence. However, longer running tasks with high data interdependence such as LU and CG are more affected by Quikolo, even amortized over 10 minutes.

$$\Delta f, i, m) = \left\{ \begin{array}{ll} 1 & : F_m(f, i) <= s \\ 0 & : F_m(f, i) > s \end{array} \right\} \tag{7.4}$$

While we compare change from one set of feature readings to the next in our algorithm, the appropriate way of testing the accuracy of our characterization is to compare change between statistics at time t and the statistics at time T. Unfortunately, this would require us to collect features until time T, so absolute accuracy metrics are not available online without sacrificing the cost saving from stopping collection early. We compare similarity and accuracy at time t and at time T in Figure 7.4(c).

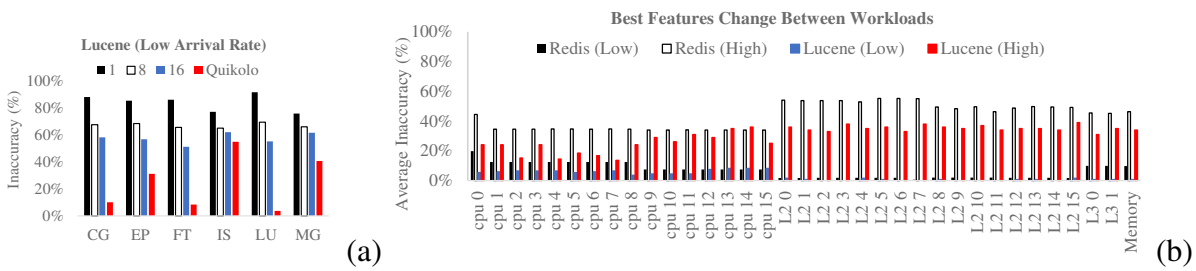

Figure 7.6: **Feature study:** (a) An increased number of collected features improves the accuracy of SLO violation identification.
(b) The features which contribute most to accuracy of SLO violation identification change between workloads.

Which Statistics Converge

In addition to studying increase in accuracy as collection time increases, we considered which statistics to use in our algorithm, such as average, standard deviation, and percentiles. Figure 7.5(a) shows the continuously updated percent difference for several statistics calculated for the L2 cache belonging to CPU 0. We found that percentiles were less susceptible to outlier readings, whereas outlier readings changed standard deviation by approximately 15% much after other statistics had converged. The statistics calculated from the full traces are shown in Figure 7.5(b). For L2 cache, L3 cache, and memory, the standard deviation was reflected by the 25th and 75th percentiles, but standard deviation added information regarding CPU utilization.

7.5 Features Study

Our implementation of Quikolo allows us to compare the accuracy of the colocation decision based on the number of features selected for use in our machine learning. We use neural networks to quantify the effect of features collected on the online characterization accuracy. For each neural network in the offline ensemble, we systematically choose some k features as inputs, and train on 80% of the workload latency information we acquired. We average our collective findings to determine which features contribute most. There are two key observations that allow this optimization to work.

(1) Not all workloads have average latencies less than the feature reading interval. In the case where the latency of a request exceeds the feature reading interval, this latency must be amortized over every time t that features were read while it was in-system. There are two separate designs for determining how a feature may impact the characterization of a workload. First, it is possible to design a neural network that uses features to directly learn expected 99th percentile latencies of a workload with latencies less than the feature reading interval. Second, it is possible to leverage knowledge of a workload's SLO violation threshold in order to simplify our neural network's output. Given a correlation between the latency of a request and the feature readings that were taken while it was present in system, we identify feature readings where an SLO violation took place. In our work, we took the latter approach.

(2) When the same number of features, starting weights, and training set are used in each neural network, the difference that would lead one neural network to be better than another

is purely the input features. With this in mind, we identify features that contribute most by averaging the inaccuracy for each neural network in which that feature was included.

Using these two observations, we compare neural networks with the following features.

- *Single feature model* explores characterization accuracy for each of our 39 features individually.
- *8 feature model* randomly chooses 8 features to use in each of 39^2 perceptrons.
- *16 feature model* randomly chooses 16 features to use in each of 39^2 perceptrons.
- *Quikolo* implements our full design for machine learning, using up to 39 features and a continually changing set of features chosen for high accuracy and timely statistical convergence.

In this experiment, we explore the relationship between dimensionality (i.e., number of features analyzed) and characterization accuracy. For this experiment, we compared the inaccuracy of the above approaches to Quikolo. While individual models can be highly accurate (within 10%), Quikolo is the only approach which is consistently accurate across multiple colocation environments. Our results for Lucene under low arrival rate are shown in Figure 7.6(a).

7.5.1 Which Features Matter

Secondly, we study which features matter across multiple workloads. For each feature, we average the inaccuracy incurred by each neural network which used that feature. We found that L2 cache miss rate was useful in identifying SLO violations incurred during low arrival rate by both Lucene and Redis. CPU utilization was useful in identifying SLO

violations by Redis under high arrival rate, but L3 and Memory usage were more useful in identifying SLO violations for Redis under low arrival rate than CPU utilization. However, Figure 7.6(b) also shows that a change in arrival rate was sufficient to change the features which contributed most to low inaccuracy. It is important to determine which features matter because pruning features that do not assist in the decision making process allows us to collect less data online and provide faster analysis to the client. In the best case, if the features that matter to a workload all come from the PID-specific /proc source, eliminating the other features reduces data collection by 78%.

7.6 Related Work

Several fields are intrinsically tied to this work, including resource management, workload characterization, and workload matriculation. The most predominant recent works in the field of resource management include Agile [111], Quasar [25], and Bolt [27]. Recent papers in workload characterization have studied Khan et al. [76], Markovian Arrival Processes [116], and Eco [60]. Workload migration is important because the machine that a service is initially scheduled onto may not be sufficient to fulfill its SLO as time passes. Elastic scheduling may also solve this problem, but it may be more effective to move a workload than add an instance of parallelism. Lastly, recent research such as [18] has used approximation for machine learning.

Resource Management

Agile is an elastic cloud-scheduling algorithm which characterizes a workload according to the amount of resource contention that the workload can support before SLO violations

are incurred. Agile's method takes 10 minutes to characterize and build a new linear regression model for any new workload, but can choose dynamically at runtime which linear regression model to use from its stable [111]. Quasar is a dynamic scheduling platform which uses fixed, short characterization runs on two architectures and in two colocation environments to extrapolate behavior in other colocation environments [25]. Bolt uses online data mining techniques to detect type and characteristics of cloud workloads. Bolt uses 10 features and uses 2-5 seconds of characterization periodically to ensure the colocation environment is still viable [27].

Workload characterization

Khan et al. apply a multiple time series approach to analyze clusters of virtual machines [76]. Eco is a daemon that uses workload characterization with the NAS Parallel Benchmarks to control performance via DVFS [60]. Pacheo-Sanchez et al. explores the Markovian Arrival Processes and queuing model for characterizing workloads in order to predict quality of service [116].

Workload migration

Enacloud encapsulates workloads into virtual machines and then dynamically places them via an energy-aware bin packing heuristic [89]. Voorsluys et al. analyze the effects of live migration on the migrating workloads [160]. CloudNet is a system that allows live migration not just across machines in the same data center, but across multiple geolocated data centers [167]. Autoscaling requires precise resource usage estimation. Roy et al. attacks the problem of workload forecasting using a model-predictive algorithm. [127]. CloudSim

is a toolkit for modeling and simulating a cloud computing environment. Their case study includes dynamic workload placement [17]. Ward et al. augments the Darwin framework with extensions for automatic migration of business critical workloads to the cloud [162]. Ye et al. analyzes different resource reservation methods for live migration [171]. While live migration is very relevant to our work, we instead create a live duplicate of a workload while minimizing our effect on the original deployment.

While previous work in resource management has used workload characterization to determine when to migrate workloads, in this chapter we empirically study the effects of characterization duration and number of features on the accuracy of decisions regarding workload migration.

Chapter 8: Conclusion

I trade accuracy to reduce overhead when characterizing online, data-intensive workloads for cloud resource allocation. Figure 8.1 describes my work in terms of offline profiling, online service analysis, and online colocation characterization. Offline profiling allows us to make improvements to online performance of workloads, but at the cost of additional resources which do not contribute to revenue. Online characterization allows resources to be added to a service only when it needs them, but at the cost of slowdown on the characterized service. Online characterization also impacts any workloads colocated with the characterized service, but limiting the resources used and identifying needed features helps reduce this overhead with limited effect on accuracy. The following paragraphs describe outcomes and conclusions from the thesis, by chapter.

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 Chapter 2, 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,

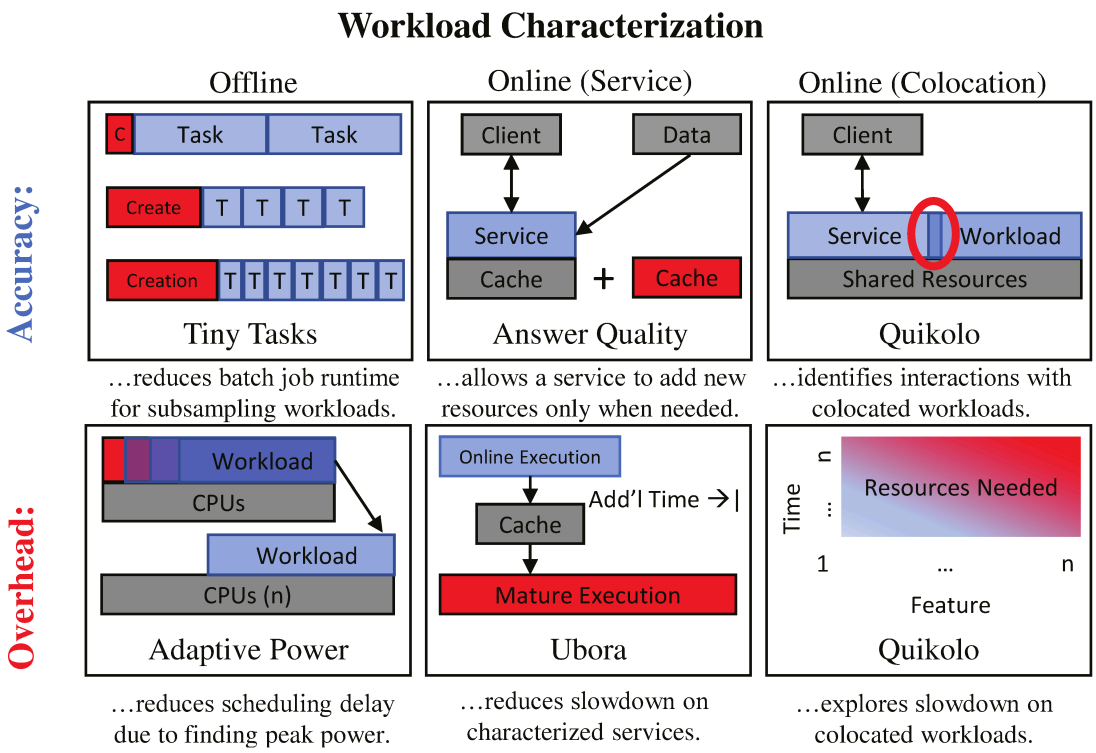

Figure 8.1: Workload characterization increases accuracy and overhead in online cloud resource allocation.

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 implemented our approach on the BashReduce platform, which is lightweight in comparison to Hadoop. We tested our approach on subsampling workloads such as genetic analysis and e-commerce datasets, which are characterized by random access patterns within the given task. Under default parameters, our approach performed 23% better

than BashReduce with the default scheduler. When outlier data accesses (time > average) were omitted, our performance increase reduced to 15%. We used these short, interactive workloads to compare our BashReduce platform to Hadoop. Our improved BashReduce platform performed 9X better than vanilla Hadoop.

In Chapter 3, we analyzed power traces from profiling core scaling on multiple high performance computing benchmarks over 3 multicore Intel architectures. We found that peak power exhibited up to 30% variation between workloads on the same architecture. We saw that a workload which was a good predictor of peak power for a target workload was not guaranteed to be a good predictor of peak power for that same workload on another architecture. We show that profiling for $k\%$ of a target workload results in variable accuracy across workloads. However, $k\%$ profiling set at 40% resulted in less than 5% inaccuracy for most workloads. Our key insight was that for the same workload on the same architecture, the traces of peak power showed similar phases occurring near the same points in the normalized runtime between different numbers of active cores. We developed an algorithm that used this key insight to reduce the amount of time $k\%$ profiling needed to attain approximate peak power profiles across multiple core counts for batch HPC workloads. Using this method, we could save up to 93% profiling time on the Intel Xeon Phi architecture with 0.03% increase in median inaccuracy. On the Intel i7 architecture, we saved an average of 11% profiling time with 1.5% increase in median inaccuracy for the NAS Parallel benchmarks and our high performance computing kernels.

Bandwidth and latency for datacenter networks has grown much faster than for disks. Emerging 40Gb/E and hybrid electrical and optical switches suggest that this trend will continue. Because it is and for the near future will be a faster solution than local disk,

networked in-memory storage is likely to underlie many data processing platforms going forward. However, networked storage suffers from the well-known, widespread problem of access times with heavy tail distributions. Chapter 4 quantifies the effect of outliers on processes that rely on the map reduce model. These heavy tail outliers slow down the data processing pipeline at the mapping stage, and when they happen close enough to the beginning of a map, they cannot be easily masked. We saw that workloads with short map times and large data sets were most affected, with delays up to 70%. We created a model predicting the effect of these outliers in order to assess one possible solution to heavy tails: replication for predictability. This approach, usable for read-only workloads, masks outliers by redundantly sending accesses to multiple nodes containing the same data and taking the response from the first. Our model used only 5% of storage capacity for replication for predictability, yet we often reduced slowdown by more than 7% using this approach.

OLDI queries have complex and data-parallel execution paths that must produce results quickly. Data used by each query is skewed across data partitions, causing some queries to time out and return premature results. Chapter 5 describes answer quality in the context of Natural Language Processing services, such as question answering and search engines. Answer quality is a metric that assesses the impact of timeouts on the quality of results. It is challenging to compute online because it requires results from mature executions that are unaffected by timeouts. Chapter 5 uses offline answer quality analysis to explore cache provisioning policies, including provisioning more cache on loss of answer quality.

Chapter 6 describes Ubora, a design approach to speed up mature executions by reusing intermediate computations from online queries, i.e., memoization. Ubora adopts a challenging systems-level approach that allows us to measure answer quality for a wide range

of services. Our implementation assumes these OLDI services use TCP connections between components, and that a single message from a calling component will result in a stream of data sent from the called component. Our implementation includes novel context tracking for commodity operating systems and bandwidth optimizations. The evaluation shows that Ubora produces mature results faster than competing transparent approaches and nearly as fast as a less flexible, application-specific approach.

We have evaluated Ubora on Apache Lucene with Wikipedia data, OpenEphyra with New York Times data, EasyRec recommendation engine with Netflix data and Hadoop/Yarn with BigBench data [153, 166, 130, 150, 125, 110, 152, 63]. Ubora slows down normal query executions by less than 7% on average. Ubora completes mature executions almost as quickly as query tagging, which eschews transparency for efficiency, with slowdown ranging from 8–16%. We also compared Ubora to timeout toggling, an alternative approach that does not require changing application source code if allowed processing time is a configuration setting for the application. However, under this approach all currently executing queries operate under the same context. Ubora exhibited a 7X speedup in finishing mature executions over timeout toggling.

Most importantly, Ubora produces answer quality quickly enough to enhance online system management. We used Ubora to guide online management, increasing throughput compared to offline approaches. We adaptively shed low priority queries to our Apache Lucene and EasyRec systems. The goal was to maintain high answer quality for high priority queries. Ubora provided answer quality measurements quickly enough to detect shifts in the arrival rate and query mix. The other transparent approach to measure answer quality,

i.e., toggling timeouts, produced mature executions too slowly. This approach allowed answer quality to fall below 90% 12X much more often than Ubora. We also used component timeouts as a proxy for answer quality [66]. This metric is available after online executions without conducting additional mature executions. As a result, it has much lower overhead. However, component timeouts are a conservative approximation of answer quality because they do not assess the effect of timeouts on answers. While achieving the same answer quality on high priority queries, Ubora-driven admission control improved peak throughput on low priority queries by 55% compared to admission control powered by component timeouts.

We also studied the predictive power of hardware counters to answer quality on Redis, a key value store we used with the Lucene workload [122, 153]. Predictive hardware counters enable preemptive actions, e.g., extending timeouts before they are triggered. We counted level-1 cache (L1) misses, level-2 cache (L2) misses, and translation lookaside buffer (TLB) misses during periods with high (>90%) and low answer quality. After executing 10% of a query, L2 misses for Redis were good predictors of low-quality answers. However, their predictive power varied across components.

We believe that the transparent design of Ubora can be of use to future frameworks aiming to share context among a cluster of machines. Custom, hand coded approaches could possibly achieve similar gains but Ubora can help a wide range of multi-component services including outreach efforts, as in [105]. For instance, we have used Ubora to dynamically tune cache size in the OpenEphyra question answering system to support Science, Technology, Engineering, and Mathematics outreach. We developed a unit to teach big data and natural language processing using Ubora to facilitate a classroom game where students

compete against an online question answering service. By dynamically allocating or reducing cache size to match its competitors' knowledge base, we hope that the Open Ephyra question answering system will be able to adequately compete with people of multiple age ranges across a broad range of knowledge categories. Our conclusion is that Ubora democratizes answer quality, allowing many services to provide high quality results and fast response times.

In chapter 7, we presented a design for a speculative deployment engine that enables in-situ workload characterization in a colocation environment. Our design assumes that the cloud service does not offer incentives to services launching in new colocation environments, as Amazon EC2 does. Our implementation of this design, Quikolo, deploys a client workload as a pod in a Kubernetes and Docker environment, then uses the /proc interface and Intel PCM hardware counters to track process-level and machine-level features twice a second. As colocation environments such as Microsoft Azure and Google Cloud Platform charge by the minute, Quikolo characterizes a workload for at least a minute and at maximum the user-supplied budget. Quikolo allowed us to study the effects of reducing the amount of time used to characterize a workload. Any additional time left in the budget is saved by the user of our service. We found that using statistical convergence as a proxy for characterization accuracy led to stopping the characterization within 3 seconds of accuracy convergence for Lucene under low arrival rate. Additional work could study the use of additional time left in budget to explore additional colocation environments. Quikolo also allowed us to study the effect of using an increased number of features on accuracy. We found that while increasing the number of features did increase the accuracy of SLO violation prediction, it was also important to track which specific features were most relevant per each workload.

Bibliography

[1] L. Abraham, V. Borkar, D. Merl, S. Subramanian, J. Allen, B. Chopra, J. Metzler, J. L. Wiener, O. Barykin, C. Gerea, D. Reiss, and O. Zed. Scuba: Diving into data at facebook. In *VLDB*, 2013.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *ASPLOS*, 2012.

[3] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Modeling and exploiting query interactions in database systems. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, 2008.

[4] Inc. Amazon Web Services. Amazon elastic compute cloud. `http://aws.amazon.com/ec2/pricing/`.

[5] M.T.A. Amin, S. Li, M. R. Rahman, P. T. Seetharamu, S. Wang, T. Abdelzaher, I. Gupta, M. Srivatsa, R. Ganti, R. Ahmed, and H. Le. Socialtrove: A self-summarizing storage service for social sensing. In *IEEE ICAC*, 2015.

[6] Cristiana Amza, Gokul Soundararajan, and Emmanuel Cecchet. Transparent caching with strong consistency in dynamic content web sites. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 264–273. ACM, 2005.

[7] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *USENIX OSDI*, 2010.

[8] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *USENIX OSDI*, 2012.

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

[10] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[11] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The nas parallel benchmarks. In *RNR*, 1994.

[12] Oleksandr Barykin, Bhuwan Chopra, Ciprian Gerea, Josh Metzler, Subbu Subramanian, Janet Wiener, David Reiss, and Daniel Merl. Scuba: Diving into Data at Facebook. *International Conference on Very Large Data Bases (VLDB)*, 2013.

[13] Leonardo Bautista-Gomez, Ana Gainaru, Swann Perarnau, Devesh Tiwari, Saurabh Gupta, Christian Engelmann, Franck Cappello, and Marc Snir. Reducing waste in extreme scale systems through introspective analysis. In *IEEE IPDPS*, 2016.

[14] R. Bertran, M. Gonzelez, X. Martorell, N. Navarro, and E. Ayguade. A systematic methodology to generate decomposable and responsive power models for cmps. In *IEEE Transactions on Computers*, 2012.

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

[16] S. Bouchenak. Automated control for sla-aware elastic clouds. In *FeBid*, 2010.

[17] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[18] Aniket Chakrabarti, Bortik Bandyopadhyay, and Srinivasan Parthasarathy. Improving locality sensitive hashing based similarity search and estimation for kernels. In *European Conference on Machine Learning*, 2016.

[19] Yu Chen. Detecting web page structure for adaptive viewing on small form factor devices. In *WWW*, 2003.

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

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

[22] J. Dean. Achieving rapid response times in large online services, 2012.

[23] J. Dean and L. Barroso. The tail at scale. In *Communications of the ACM*, 2013.

[24] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.

[25] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ASPLOS*, 2014.

[26] Christina Delimitrou, Nick Bambos, and Christos Kozyrakis. Qos-aware admission control in heterogeneous datacenters. In *IEEE ICAC*, 2013.

[27] Christina Delimitrou and Christos Kozyrakis. Bolt: I know what you did last summer... in the cloud. In *ASPLOS*, 2017.

[28] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, 2003.

[29] Dormando. Memcached: A distributed memory object caching system. `www.memcached.org`.

[30] Hadi Esmaeilzadeh, Emily Blem, Renee Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[31] M. Etinski, J. Corbalan, J. Labarta, and M. Valero. Optimizing job performance under a given power constraint in hpc centers. In *International Green Computing Conference*, 2010.

[32] R. Falsett, R. Seyer, and C. Siemers. Limitation of the response time of a software process, December 29 2004. WO Patent App. PCT/EP2003/000,721.

[33] W. Felter, K. Rajamani, T. Keller, and C. Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *SUPERCOMPUTING*, 2005.

[34] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. Murdock, E. Hyberg, J. Prager, N. Schlaerfer, and C. Welty. The ai behind watson—the technical article. In *The AI Magazine*, 2010.

[35] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *USENIX NSDI*, 2007.

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

[37] B. Forrest. Bing and google agree: Slow pages lose users. radar.oreilly.com, 2009.

[38] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Adapting to network and client variation using infrastructural process proxies: lessons and perspectives. *Personal Communications*, 5:10–19, 1998.

[39] Erik Frey. bashreduce : mapreduce in a bash script.

[40] X. Fu, X. Wang, and C. Lefurgy. How much power oversubscription is safe and allowed in data centers. In *IEEE ICAC*, 2011.

[41] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, , and Li Zhang. Adaptive, model-driven autoscaling for cloud applications. In *International Conference on Autonomic Computing*, 2014.

[42] J. Gantz and D. Reinsel. Extracting value from chaos. In *IDC*, 2011.

[43] A. Gelfond. Tripadvisor architecture - 40m visitors, 200m dynamic page views, 30tb data. http://highscalability.com, June 2011.

[44] H. Ghasemi and N. Kim. Rcs: runtime resource and core scaling for power-constrained multi-core processors. In *PACT*, 2014.

[45] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H. Jacobsen. Bigbench: Towards an industry standard benchmark for big data analytics. In *ACM SIGMOD*, 2013.

[46] I. Goiri, R. Bianchini, S Nagarakatte, and T Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ACM ASPLOS*, 2015.

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

[48] I. 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.

[49] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX OSDI*, 2012.

[50] Google. Google cloud platform pricing calculator - google cloud platform. https://cloud.google.com/products/calculator/, 2015.

[51] Y. Guo, P. Lama, J. Rao, and X. Zhou. V-cache: Towards flexible resource provisioning for multi-tier applications in iaas clouds. In *International Symposium on Parallel and Distributed Processing*, 2013.

[52] Saurabh Gupta, Devesh Tiwari, Christopher Jantzi, James Rogers, and Don Maxwell. Understanding and exploiting spatial properties of system failures on extreme-scale hpc systems. In *Int'l Conference on Dependable Systems and Networks (DSN)*, 2015.

[53] J. Hardman. Nas parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html, 2012.

[54] Yuxiong He, Sameh Elnikety, James Larus, and Chenyu Yan. Zeta: Scheduling interactive services with partial execution. In *ACM SOCC*, 2012.

[55] Yuxiong He, Sameh Elnikety, and Hongyang Sun. Tians scheduling: Using partial processing in best-effort applications. In *ICDCS*, 2011.

[56] Yuxiong He, Zihao Ye, Qiang Fu, and Sameh Elnikety. Budget-based control for interactive services with adaptive execution. In *IEEE ICAC*, 2012.

[57] Henry Hoffmann and Martina Maggio. Pcp: A generalized approach to optimizing performance under power constraints through resource management. In *International Conference on Autonomic Computing*, 2014.

[58] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin C. Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2001.

[59] S. Hsiao, L. Massa, and V. Luu. An epic tripadvisor update: Why not run on the cloud? the grand experiment. http://highscalability.com, October 2012.

[60] S Huang and W Feng. Energy-efficient cluster computing via accurate workload characterization. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 68–75. IEEE Computer Society, 2009.

[61] Jinho Hwang and Timothy Wood. Adaptive performance-aware distributed memory caching. In *IEEE ICAC*, 2013.

[62] Intel. Intel xeon phi coprocessor. http://www.colfax-intl.com/nd/downloads/Xeon-Phi-Coprocessor-Datasheet.pdf, 2014.

[63] Intel Corporation. Github - intel-hadoop/big-data-benchmark-for-big-bench: Big bench workload development. https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench, 2016.

[64] International Technology Roadmap for Semiconductors. The itrs dram cost is the cost per bit (packaged microcents) at production. http://www.itrs.net/.

[65] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys Conf.*, 2007.

[66] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *SIGCOMM*, 2013.

[67] M. Jeon, Y. He, S. Elnikety, A. Cox, and S. Rixner. Adaptive parallelization of web search. In *EuroSys Conf.*, 2013.

[68] Niranjan Kamat, Prasanth Jayachandran, Kathik Tunga, and Arnab Nandi. Distributed interactive cube exploration. In *ICDE*, 2014.

[69] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *SOCC*, 2012.

[70] J. Kelley, C. Stewart, S. Elnikety, and Y. He. Cache provisioning for interactive nlp services. In *Workshop on Large-Scale Distributed Systems and Middleware*, 2013.

[71] Jaimie Kelley and Christopher Stewart. Balanced and predictable networked storage. In *International Workshop on Data Center Performance*, 2013.

[72] Jaimie Kelley, Christopher Stewart, Nathaniel Morris, Devesh Tiwari, Yuxiong He, and Sameh Elnikety. Measuring and managing answer quality for online data-intensive services. In *International Conference on Autonomic Computing*, 2015.

[73] Jaimie Kelley, Christopher Stewart, Devesh Tiwari, Sameh Elnikety, Yuxiong He, and Nathaniel Morris. Open-source benchmarks for online data-intensive services (tutorial). http://web.cse.ohio-state.edu/~kelley.530/ubora/tutorial.html, 2015.

[74] Jaimie Kelley, Christopher Stewart, Devesh Tiwari, and Saurabh Gupta. Adaptive power profiling for many-core hpc architectures. In *International Conference on Autonomic Computing*, 2016.

[75] J. Kephart and J. Lenchner. A symbiotic cognitive computing perspective on autonomic computing. In *IEEE ICAC*, 2015.

[76] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1287–1294. IEEE, 2012.

[77] 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.

[78] M. Kicherer. anyc / librapl github. `http://github.com/anyc/librapl`, 2013.

[79] T. Kidd. Intel xeon phi coprocessor power management configuration: Using the micsmc command-line interface. `https://software.intel.com/en-us/blogs/2014/01/31/intel-xeon-phi-coprocessor-power-management-configuration-using-the-micsmc-`, 2014.

[80] T. Komoda, S. Hayashi, T. Nakada, S. Miwa, and H. Nakamura. Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping. In *ICCD*, 2013.

[81] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *ACM SIGMOD*, 2012.

[82] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *USENIX OSDI*, 2012.

[83] Palden Lama and Xiaobo Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, IEEE ICAC, 2012.

[84] Lawrence Livermore National Security. Minife summary v 2.0. `https://asc.llnl.gov/CORAL-benchmarks/Summaries/MiniFE_Summary_v2.0.pdf`, 2014.

[85] George Lawton. LAMP lights enterprise development efforts. *Computer*, 9:18–20, 2005.

[86] G. Lee, N. Tolia, P. Ranganathan, and R. Katz. Topology-aware resource allocation for data-intensive workloads. In *APSys*, 2010.

[87] Jon Lenchner. Knowing what it knows: selected nuances of watson's strategy. `http://ibmresearchnews.blogspot.com`, 2011.

[88] J. Levon, P. Elie, and M. Johnson. Oprofile - a system profiler for linux. `http://oprofile.sourceforge.net/`.

[89] Bo Li, Jianxin Li, Jinpeng Huai, Tianyu Wo, Qin Li, and Liang Zhong. Enacloud: An energy-saving application live placement approach for cloud computing environments. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 17–24. IEEE, 2009.

[90] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *ACM SOSP*, Cascais, Portugal, October 2011.

[91] Lucid Imagination. The case for lucene/solr: Real world search applications. White Paper, 2008.

[92] 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.

[93] Steve Mackie. How fast is our data volume growing. Storage Strategies Inc., 2009.

[94] H. Madhyastha, J. McCullough, G Porter, R Kapoor, S Savage, A. Snoeren, and A Vahdat. scc: Cluster storage provisioning informed by application characteristics and slas. In *FAST*, 2012.

[95] Christopher Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[96] F. McSherry, D. G. Murray, R. Issacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[97] D. Meisner, B. Gold, and T. Wenisch. Powernap: Eliminating server idle power. In *ACM ASPLOS*, March 2009.

[98] D. Meisner, C. Sadler, L. Barroso, W-D. Weber, and T. F. Wenisch. Power management of on-line data intensive services. In *ISCA*, 2011.

[99] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, and M. Tolton adn T. Vassilakis. Dremel: Interactive analysis of web-scale dataasets. In *VLDB*, 2010.

[100] Daniel Menasce. Workload characterization. https://cs.gmu.edu/ menasce/cs672/slides/CS672-wkldchar.pdf, 1999.

[101] Daniel Menasce. Workload characterization. In *IEEE Internet Computing*, 2003.

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

[103] R.W. Moore and B.R. Childers. Using utility prediction models to dynamically choose program thread counts. In *IEEE Int. Symp. Performance Analysis of Systems Software*, 2012.

[104] Nathaniel Morris, Siva Meenakshi Renganathan, Christopher Stewart, Robert Birke, and Lydia Chen. Sprint ability: How well does your software exploit bursts in processing capacity? In *International Conference on Autonomic Computing*, 2016.

[105] Stephanie Muhammad, Jaimie Kelley, and Christopher Stewart. Ed watson: Teaching big data to k-12 students. *2016 Spring Undergraduate Research Expo*, 2016.

[106] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[107] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In *USENIX NSDI*, 2011.

[108] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *IEEE 27th International Conference on Data Engineering*, 2011.

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

[110] Netflix. Netflix prize. `http://www.netflixprize.com/`, 2009.

[111] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *International Conference on Autonomic Computing*, 2013.

[112] Bin Nie, Devesh Tiwari, Saurabh Gupta, Evgenia Smirni, and James H. Rogers. A large-scale study of soft-errors on gpus in the field. In *IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016*, 2016.

[113] Cynthia Nottingham. Linux vm sizes microsoft azure. `https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-sizes/`, 2016.

[114] 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.

[115] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Scalable scheduling for sub-second parallel jobs. In *SOSP*, 2013.

[116] Sergio Pacheco-Sanchez, Giuliano Casale, Bryan Scotney, Sally McClean, Gerard Parr, and Stephen Dawson. Markovian workload characterization for qos prediction in the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 147–154. IEEE, 2011.

[117] Joao Paiva, Pedro Ruivo, Paolo Romano, and Luís Rodrigues. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. In *International Conference on Autonomic Computing*, 2013.

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

[119] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *USENIX OSDI*, 2010.

[120] A. Rasmussen, M. Conley, R. Kapoor, V. Lam, G. Porter, and A. Vahdat. Themis: An i/o-efficient mapreduce. In *ACM SOCC*, 2012.

[121] A. Rasmussen, G. Porter, M. Conley, G. Madhyastha, R. Mysore, A. Pucher, and A. Vahdat. Tritonsort: A balanced large-scale sorting system. In *USENIX NSDI*, 2012.

[122] Redislabs. Redis. `http://redis.io/`, 2016.

[123] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn McKinley. Exploiting processor heterogeneity in interactive services. In *IEEE ICAC*, 2013.

[124] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn S McKinley. Exploiting processor heterogeneity in interactive services. In *ICAC*, pages 45–58, 2013.

[125] Research Studios Austria Forschungsgesellschaft mbH. Easyrec-open source recommendation engine. `http://easyrec.org/`, 2014.

[126] C. Roe. The growth of unstructured data: What to do with all those zettabytes? `www.dataversity.net`, 2012.

[127] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507. IEEE, 2011.

[128] O. Sarood, A. Langer, A. Gupta, and L. Kale. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *IEEE Supercomputing*, 2014.

[129] M. C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[130] Nico Schlaefer. The ephyra question answering system. `https://sourceforge.net/projects/openephyra/`, 2013.

[131] R. Schone, D. Hackenberg, and D. Molka. Memory performance at reduced cpu clock speeds: An analysis of current x86 64 processors. In *HOTPOWER*, 2014.

[132] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: An os facility for fine-grained power and energy management on multicore servers. In *ACM ASPLOS*, 2012.

[133] D. Shue, M. Freedman, and A. Shaikh. Performance isolation and fairness for multitenant cloud storage. In *USENIX OSDI*, 2012.

[134] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbag. Dapper, a large-scale distributed systems tracing infrastructure. In *Google Technical Report*, 2010.

[135] Michael Sindelar, Ramesh K Sitaraman, and Prashant Shenoy. Sharing-aware algorithms for virtual machine colocation. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 367–378. ACM, 2011.

[136] SINTEF. Big data, for better or worse: 90% of world's data generated over last two years. http://www.sciencedaily.com/releases/2013/05/130522085217.htm, May 2013.

[137] R. Smith. Intel's knights landing co-processor detailed. http://www.anandtech.com, 2014.

[138] Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. Rpc chains: Efficient client-server communication in geodistributed systems. In *USENIX NSDI*, 2009.

[139] Simon Spinner, Giuliano Casale, Xiaoyun Zhu, and Samuel Kounev. Librede: A library for resource demand estimation. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, 2014.

[140] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *EuroSys Conf.*, March 2007.

[141] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *USENIX NSDI*, May 2005.

[142] C. Stewart and K. Shen. Some joules are more precious than others: Managing renewable energy in the datacenter. In *Workshop on Power Aware Computing and Systems(HotPower)*, September 2009.

[143] C. Stewart, K. Shen, A. Iyengar, and J. Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *IEEE MASCOTS*, 2010.

[144] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *ICAC*, 2013.

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

[146] M. Stokely, A. Mehrabian, C. Albrecht, F. Labelle, and A. Merchant. Projecting disk usage based on historical trends in a cloud environment. In *ScienceCloud*, 2012.

[147] M. Suleman, M. Quresh, and Y.N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. In *ACM ASPLOS*, 2008.

[148] T. Anome et al. Wikipedia:modelling wikipedia's growth. https://en.wikipedia.org/wiki/Wikipedia:Modelling_Wikipedia's_growth, 2014.

[149] Kun Tang, Devesh Tiwari, Saurabh Gupta, Ping Huang, Qiqi Lu, Christian Engelmann, and Xubin He. Power-capping aware checkpointing: On the interplay among power-capping, temperature, reliability, performance, and energy. In *Int'l Conference on Dependable Systems and Networks (DSN)*, 2016.

[150] Technology Laboratory's (ITL) Retrieval Group. Text retrieval conference data. http://trec.nist.gov/data.html, 2014.

[151] The Apache Software Foundation. Welcome to apache hadoop. hadoop.apache.org.

[152] The Apache Software Foundation. Apache hadoop 2.7.1 - apache hadoop nextgen mapreduce (yarn). https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html, 2015.

[153] The Apache Sofware Foundation. Apache lucene. http://lucene.apache.org/core/, 2016.

[154] Devesh Tiwari, Saurabh Gupta, George Gallarno, Jim Rogers, and Don Maxwell. Reliability lessons learned from gpu experience with the titan supercomputer at oak ridge leadership computing facility. In *Supercomputing (SC)*, 2015.

[155] Devesh Tiwari et al. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[156] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *The First International Workshop on Parallel Software Tools and Tool Infrastructures*, 2010.

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

[158] Vernon Turner, David Reinsel, F. John Gantz, and Stephen Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *IDC Report*, 2014.

[159] Evangelos Vlachos, Michelle L Goodstein, Michael A Kozuch, Shimin Chen, Babak Falsafi, Phillip B Gibbons, and Todd C Mowry. Paralog: Enabling and accelerating

online parallel monitoring of multithreaded applications. *ACM SIGARCH Computer Architecture News*, 38(1):271–284, 2010.

[160] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *IEEE International Conference on Cloud Computing*, pages 254–265. Springer, 2009.

[161] Shinan Wang, Bing Luo, Weisong Shi, and Devesh Tiwari. Application configuration selection for energy-efficient execution on multicore systems. *Journal of Parallel and Distributed Computing*, 87:43–54, 2016.

[162] Christopher Ward, N Aravamudan, Kamal Bhattacharya, Karen Cheng, Robert Filepp, R Kearney, B Peterson, Larisa Shwartz, and Christopher C Young. Workload migration into clouds challenges, experiences, opportunities. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 164–171. IEEE, 2010.

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

[164] T. White. *Hadoop: The definitive guide*. O'Reilly Media, Sebastopol, CA, 2012.

[165] A. Wierman, Z. Liu, I. Liu, and H. Mohsenian-Rad. Opportunities and challenges for data center demand response. In *IEEE IGCC*, 2014.

[166] Wikimedia Foundation. Wikimedia downloads. `https://dumps.wikipedia.org/`, 2014.

[167] Timothy Wood, KK Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines. In *ACM Sigplan Notices*, volume 46, pages 121–132. ACM, 2011.

[168] J. Xie, S. Yin, X. Ruan, Z. Ding, J. Majors, and X. Qin. Improving mapreduce performance via data placement in heterogeneous hadoop clusters. In *International Heterogeneity in Computing Workshop*, 2010.

[169] Zichen Xu, Nan Deng, Christopher Stewart, and Xiaorui Wang. Cadre: Carbon-aware data replication for geo-diverse services. In *International Conference on Autonomic Computing*, 2015.

[170] D. Yang and C. Stewart. Zoolander: Modelling and managing replication for predictability. In *Technical Report, The Ohio State University*, 2011.

[171] Kejiang Ye, Xiaohong Jiang, Dawei Huang, Jianhai Chen, and Bei Wang. Live migration of multiple virtual machines with resource reservation in cloud computing environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 267–274. IEEE, 2011.

[172] 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 *NSDI*, 2012.

[173] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *ASPLOS*, 2016.

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

[175] Z. Zhang, L. Cherkasova, A. Verma, and B. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *IEEE ICAC*, September 2012.

[176] Y. Zheng, B. Ji, N. Shroff, and P. Sinha. Forget the deadline: Scheduling interactive applications in data centers. In *CLOUD*, 2015.

[177] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3), 1996.