scc: Cluster Storage Provisioning Informed by Application Characteristics and SLAs

Harsha V. Madhyastha[*], *John C. McCullough*[†], *George Porter*[†], *Rishi Kapoor*[†],
Stefan Savage[†], *Alex C. Snoeren*[†], *and Amin Vahdat*[†]
UC Riverside[*] and UC San Diego[†]

Abstract

Storage for cluster applications is typically provisioned based on rough, qualitative characterizations of applications. Moreover, configurations are often selected based on rules of thumb and are usually homogeneous across a deployment; to handle increased load, the application is simply scaled out across additional machines and storage of the same type. As deployments grow larger and storage options (e.g., disks, SSDs, DRAM) diversify, however, current practices are becoming increasingly inefficient in trading off cost versus performance.

To enable more cost-effective deployment of cluster applications, we develop *scc*—a storage configuration compiler for cluster applications. *scc* automates cluster configuration decisions based on formal specifications of application behavior and hardware properties. We study a range of storage configurations and identify specifications that succinctly capture the trade-offs offered by different types of hardware, as well as the varying demands of application components. We apply *scc* to three representative applications and find that *scc* is expressive enough to meet application Service Level Agreements (SLAs) while delivering 2–4.5× savings in cost on average compared to simple scale-out options. *scc*'s advantage stems mainly from its ability to configure heterogeneous—rather than conventional, homogeneous—cluster architectures to optimize cost.

1 Introduction

Today, application providers can choose from a range of storage choices to provision the infrastructure for cluster-based applications. Storage technologies as diverse as DRAM, solid state drives (SSDs), and hard disks present complex trade-offs in cost, capacity, performance (along multiple dimensions), and power consumption. New storage technologies such as phase change memory [14] will soon further complicate the space.

Provisioning, however, is based largely on rules of thumb and best practices. Applications are broadly categorized as storage, compute, or memory intensive and are typically deployed on homogeneous clusters heavy on the corresponding resource. As application load increases, deployments are "scaled out" by simply adding more storage and compute in the same configuration. Not only does this state of affairs fail to take full advantage of the diversity of available storage choices, but the increasing scale of deployments makes such inefficiencies worse; inefficiencies multiplied over thousands of servers can have substantial costs. In the scale-out model, a poor initial choice can greatly inflate expenses.

In this paper, we pursue an alternate approach—the automated selection of cluster storage configurations based on formal specifications of applications, hardware, and workloads. Initially, such an approach places significant burden on those developing and deploying applications to characterize applications and workloads. However, the resultant savings in cost necessary to satisfy Service Level Agreements (SLAs) can be substantial.

Our primary contributions in implementing this approach are two-fold. First, we determine how the characteristics of applications, workloads, and hardware should be specified in order to automate the selection of cluster configurations. To do so, we study several representative deployment scenarios and identify a parsimonious yet sufficiently expressive set of parameters that capture the trade-offs offered by different types of storage devices and the varying demands across application components. Though others have pursued a similar approach of formally specifying workloads and hardware [5, 7, 34], we extend this approach to account for various types of storage media (e.g., disk, SSD, and DRAM) and to jointly capture storage and compute requirements of applications. We show that it is feasible to concisely summarize the most salient parameters that determine the resource requirements of specific application deployments, thus minimizing the burden of formal specification.

Second, we develop *scc*, a storage configuration compiler that takes specifications of applications, workloads,

Resource	MB/s	IOPS	Watts	Cost
7.2K Disk (500 GB)	90 (R) 90 (W)	125 (R) 125 (W)	5	$213
15K Disk (146 GB)	150 (R) 150 (W)	285 (R) 285 (W)	2.3	$296
SSD (32 GB)	250 (R) 80 (W)	2500 (R) 1000 (W)	2.4	$456
DRAM (1 GB)	12.8K (R) 12.8K (W)	1.6B (R) 1.6B (W)	3.5	$35
CPU core	-	-	20	$137

Server type	Resource Limits	Cost
Server1	4 cores, 1 Gbps network 12GB DRAM, 4 SAS slots	$1400
Server2	16 cores, 10 Gbps network 48GB DRAM, 16 SAS slots	$1850
Server3	32 cores, 10 Gbps network 512GB DRAM, 16 SAS slots	$11000

Table 1: Example set of cluster building blocks input to *scc*. Cost is price plus energy costs for 3 years. *scc* takes read and write *gap* parameters as input rather than IOPS.

and hardware as input, automates the navigation of the large space of storage configurations, and zeroes in on the configuration that meets application SLAs at minimum cost. To evaluate *scc*, we experiment with three distributed applications with distinctly different workload characteristics: 1) ProductSearch, a product search webservice modeled on Google Merchant Center [17], 2) Terasort, a MapReduce job to sort large tuple collections, and 3) PhotoShare, a photo-sharing Web service modeled on Flickr. By deploying these applications on a range of cluster configurations and measuring application performance on these configurations, we present empirical evidence that *scc* is expressive enough to capture the needs of a range of applications.

In developing *scc* and applying it to diverse application workloads, we make three key observations. First, the right choice of storage configuration depends not only on the storage capacity and I/O needs of the application, but also on the application's compute requirements and on the types of server configurations available. When an application performs a set of operations in sequence, the resources assigned to serve each of these operations must be jointly optimized to satisfy the performance bound on the sequence of operations at minimum cost. For example, in an application that performs an I/O operation on some data followed by some computation, the storage type assigned to the data depends on the amount of computation. When the computation consumes significant time, the data may need to be stored on fast storage like SSDs to meet performance bounds, whereas when compute time is low, there is greater slack in performing the I/O and hence, slower cheaper storage like disk may suffice.

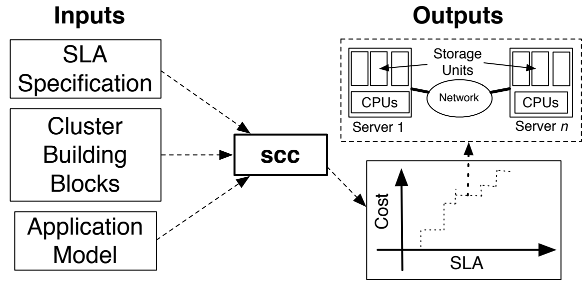


Figure 1: Overview of *scc*.

Second, we find that clusters with heterogeneity—rather than conventional homogeneity—across servers are necessary to optimize cost. The resources required differ across application components because of varying ratios of capacity, compute, and I/O throughput needs across components. For example, in a deployment of the photo-sharing Web service, it may be cheaper to store photos on disk and cache thumbnails in DRAM; storing both on disk or both in DRAM may result in higher cost due to higher I/O throughput needs from thumbnails or higher storage capacity needs of photos, respectively. As a result, *scc*'s suggested configuration meets performance SLAs at low cost. For example, in experiments with Terasort, we find that *scc* meets performance requirements at 15–20% lower cost than a homogeneous configuration recommended based on best practices.

Finally, we also find that the most cost-effective cluster architecture depends not only on the application being provisioned but also on the workload and performance requirements. Data that was initially capacity-bound may become I/O-bound at higher loads, calling for shifts from high capacity but slow storage, e.g., disks, to low capacity but fast storage, e.g., SSDs. As a result, cluster configurations output by *scc* for ProductSearch and PhotoShare result in 2x–4.5x average savings in cost compared to similarly performant scale-out options.

2 Problem setting and overview

Identifying an appropriate cluster architecture to host a large-scale service is often not straightforward. For example, given a set of resources to choose from (e.g., as shown in Table 1), an application provider has to answer several questions. What storage technologies should be employed, and how should data be partitioned across them? Where should caching be employed? What types of servers should be chosen to house the selected storage units? In addition, even if the application's implementation is efficient and there is coarse-grained parallelism in the underlying workload, how will algorithmic shifts in the application or variations in workload affect the appropriate cluster architecture? Our goal is to automate the process of answering these questions, rather than relying solely on human judgment.

Problem setting. In developing *scc*, our focus is on the typical scenario where a cluster is dedicated to a specific application, rather than large-scale data centers (e.g., Google, Microsoft) that host a mix of applications. *scc* caters to the common case where an application provider either acquires hardware or uses third-party infrastructure to deploy an application. In such cases, the question we seek to answer is: what information from the infrastructure provider and from the application developer is necessary to determine a cost-effective cluster configuration that meets performance goals?

Overview of *scc*. As shown in Figure 1, *scc* takes three inputs: i) a model of application behavior, specified by the application's developer, ii) characteristics of available hardware building blocks specified by the infrastructure provider, and iii) application performance metrics, i.e., a parameterized service level agreement (SLA). Given these inputs, *scc* computes how cluster cost varies as a function of SLA and outputs a low-cost cluster configuration that meets the SLA at each point in the space. For example, a webservice SLA might specify a peak query rate per second. For each potential SLA value (e.g., 1000 queries per second), *scc* determines a cost-effective cluster architecture capable of satisfying the SLA. *scc*'s output cost vs. SLA value distribution helps administrators decide what performance can be supported cost effectively.

Our focus in developing *scc* is to show how to systematically exploit storage diversity; i.e, select among different physical media, local and remote storage, and various caching strategies. In the future, we plan to extend *scc* to tailor network configurations and choose among CPU types. Here, we assume the cluster network can deliver uniform bandwidth between all pairs of servers [4] and do not address incast-like scenarios [27] that arise due to limited packet buffers. Instead, we assume network storage access is limited only by network adapter speeds.

```
<sla task="photoview" rate="300"> </sla>
<sla task="photoupload" rate="100"> </sla>
<sla task="tagview" rate="100"> </sla>
```

(a)

```
<resources>
  <storage_unit name="7.2KDisk" capacity="500GB" bus="SAS"
      rateR="90MBps" gapR="8ms" rateW="90MBps" gapW="8ms"
      volatile="0" price="200" power="5W"> </storage_unit>
  <storage_unit name="SSD" capacity="32GB" bus="SAS"
      rateR="250MBps" gapR="0.4ms" rateW="80MBps" gapW="1ms"
      volatile="0" price="450" power="2.4W"> </storage_unit>
  <storage_unit name="DRAM" capacity="1GB" bus="DDR3-1333"
      rateR="12.8GBps" gapR="0.6ns" rateW="12.8GBps" gapW="0.6ns"
      volatile="1" price="25" power="3.5W"> </storage_unit>
  ... additional storage units ...
  <cpu price="85" power="20W"> </cpu>
  <server name="HP DL380 G6" price="1400" cpus="4" BW="1Gbps">
      <bus name="SAS" slots="4" BW="6Gbps"> </bus>
      <bus name="DDR3-1333" slots="12" BW="21.3GBps"> </bus>
  </server>
  ... additional servers ...
</resources>
```

(b)

```
<application>
  <dataset name="tables_repository" size="150GB" persistent="1"
      remote="1"> </dataset>
  <dataset name="hot_ratingsdata" size="1.6GB" persistent="*"
      remote="0"> </dataset>
  <dataset name="cold_ratingsdata" size="6.4GB" persistent="*"
      remote="0"> </dataset>
  ... additional datasets ...
  <task name="worker" phase="exec" memory="1GB">
    <io op="R" dataset="tables_repository" record_size="800MB"
        num_records="1" blocking="0"> </io>
    ... additional I/O operations ...
    <compute time="2.2s" blocking="1"> </compute>
    <dependency task="queryprocessor" num_invocations="1"
        parallel="1" blocking="1"> </dependency>
  </task>
  <task name="queryprocessor" phase="exec" memory="200MB">
    <io op="R" dataset="hot_ratingsdata" probability="0.8"
        num_records="40K" record_size="4KB" blocking="0"> </io>
    <io op="R" dataset="cold_ratingsdata" probability="0.2"
        num_records="40K" record_size="4KB" blocking="0"> </io>
    <compute time="0.65s" blocking="1"> </compute>
  </task>
</application>
```

(c)

Figure 2: Example specifications of (a) SLAs for PhotoShare, (b) hardware resources, and (c) application behavior for a particular deployment of ProductSearch.

3 Inputs to *scc*

We now describe how we represent the three inputs to *scc*—SLA specifications, properties of cluster building blocks, and application models. Rather than model the intricate complexities of algorithms and hardware, *scc* captures aggregate high level statistics that are relevant to application and hardware scaling behavior over a broad range of scenarios. Towards this end, we identify a key set of elements that comprise each of *scc*'s inputs and the corresponding attributes required to describe these elements. Figure 2 depicts examples of *scc*'s three inputs; our implementation encodes them in XML.

3.1 Specifying SLAs

We consider throughput-based SLAs for two distinct application classes: batch and interactive; we defer supporting latency-based SLAs to future work. For batch applications, the SLA has two attributes—the job size and the required execution time, e.g., for a MapReduce job, the SLA specifies the number of records to be processed and the total run time for doing so. *scc* is more applicable for provisioning a new set of VMs for every job than for provisioning a shared cluster used for running jobs with varying I/O and compute characteristics. For interactive applications run as services, each type of request is associated with its own performance-based SLA that describes its required sustained processing rate. For example, in the case of a photo sharing Web service, the rates of photo uploads, photo views, and album views are each specified as a separate SLA. *scc*'s SLAs specify peak rather than average case throughput. We discuss

how *scc* accounts for temporal variation in Section 6.3.

3.2 Cluster building blocks

scc's second input is a characterization of the set of building blocks available for assembling the cluster. We account for three types of elements—storage units, CPU cores, and servers. To ensure our approach is not tied to the characteristics of any particular technology, we employ abstract features such as I/O bandwidth and number of processor slots as the attributes for these elements. Table 1 lists sample building blocks used in our evaluation.

3.2.1 Storage

Storage resources come in discrete units, e.g., 1 disk or 1 stick of DRAM. To differentiate between different kinds of storage technologies such as disk, SSDs and DRAM, we characterize each unit based on two properties: capacity and I/O throughput. Capacity is simply the amount of available storage measured in bytes. Representing I/O throughput is more complex; we capture it with four attributes—the average *rate* at which I/O requests are served and the average latency *gap* between serving successive I/O requests, accounting for both separately for reads and writes. The gap parameter captures overheads involved with non-sequential I/O, e.g., seeks on disks and block erasure on SSDs. We define read (write) gap for a particular storage device as the latency incurred on average between successive reads (writes) to random logical addresses on the device. The latency to serve a read (write) request for a chunk of *size* bytes is thus $(\frac{size}{rate} + gap)$. We consider gap rather than the commonly used IOPS metric because gap enables us to better capture the range of I/O performance regions from small to large records. For example, characterizing read performance on a 7.2K-RPM disk based on IOPS and rate works well for 4 KB and 10 MB reads, but fails to capture the read throughput with 200 KB reads. In our evaluation, we find that these four attributes—rate and gap for reads and writes—suffice to capture the I/O performance of multiple disk types and SSDs. Furthermore, we believe these attributes are expressive enough to capture the characteristics of phase change memory (PCM) and other emerging storage technologies.

The application-visible performance of a storage medium is also influenced by how the chosen file system places data. For example, a disk can deliver significantly higher write throughput when written to in a log format [28]. Therefore, when an application stores a dataset on a storage or file system, we measure I/O rates and gaps of each storage unit when using that system to read/write data. Further, for each storage unit, we consider two other attributes: storage persistence (i.e., whether it provides non-volatile storage) and I/O bus type (e.g., SAS vs. PCIe).

3.2.2 Servers and compute

Servers impose constraints on how storage can be packed into a physical box. For each kind of server, we consider its memory capacity as well as the properties of its I/O controllers. For each I/O controller, we consider the total number of units it can support and its maximum available I/O bandwidth. For example, a serial attached SCSI (SAS) controller permits up to 128 connected disks, yet supports a maximum I/O bandwidth of only 6 Gbps, less than the total sequential I/O throughput that can be obtained from 128 disks. Similarly, throughput for remote storage is limited by a server's network interface speed.

As our focus is on storage complexity in cluster architectures, we consider only a single CPU type, ignoring trade-offs in compute per unit power [6, 11]. Instead, we vary the number of cores per server to extract the level of parallelism needed to maximize storage utilization.

3.2.3 Costs

Finally, an additional attribute for every element in the resource specification is the amortized cost per hardware unit including both capital and operational outlays. In our current implementation, the latter subsumes energy costs, ignoring data center costs and administrator salaries, and we consider total cluster cost to be a linear sum of individual components, which may not necessarily be true for large quantities. We leave for future work discounting the growth of expenses with cluster size and accounting for increased operational costs with a higher diversity of server configurations in the cluster.

3.3 Characterizing applications

Our characterization of applications accounts for two aspects—its implementation and the workload in its planned deployment. However, unlike previous attempts at formally specifying workloads [34], simply accounting for storage capacity needs and the application's stream of I/O operations does not suffice for our purpose. Instead, to capture an application's implementation, we first ask the application's developer to describe its decomposition into compute and storage components, and the interaction between them. For example, Figure 3 depicts the components, and the interaction between them, for one of the three applications we consider later in our evaluation—a photo sharing Web service, PhotoShare. Though our approach places the onus on application developers to go through the process of formally specifying the components of their application, an application's specification is reusable across deployments. Some of the characteristics of several applications are already captured today [23, 24].

Second, we enable those who deploy an application to annotate the specification of the application's architecture with properties of the expected workload in their

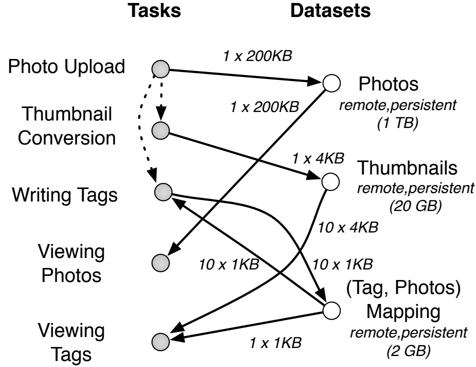


Figure 3: Interaction between tasks and datasets in example application PhotoShare. Edges between tasks and datasets represent I/O with direction differentiating input and output. Dotted edges indicate task dependencies.

deployment. To do so, we require that the compute and I/O characteristics of an application's components, when subjected to the target workload, be determined by running small-scale application benchmarks. Extracting these properties requires tracing the application's execution—now standard practice in resource-intensive performance-critical applications. In the absence of built-in tracing support, systems like Magpie [8] can be leveraged.

3.3.1 Tasks and datasets

scc's application specification separates the application's compute and storage requirements into *tasks* and *datasets*. A task is a specific application functional unit; all threads/processes that perform the same function together constitute a single task. A dataset is a collection of records of the same type with similar I/O access patterns.

Execution of tasks. To account for how compute time and I/O wait time are distributed across a task's execution, we represent each task by its *execution path*; different tasks in an application will have different execution paths. A task in an interactive application executes its execution path for each incoming request, whereas in batch applications, a task's execution path is executed as many times as necessary to consume its input. Further, since batch jobs can go through multiple phases of execution, we require the application developer to tag each task with the phase to which it belongs. The cluster can thus be provisioned to support the maximal resource requirement across phases.

We characterize the execution path of a task as a sequence of three types of operations—compute, I/O, and invocations of other tasks. Each of these can be marked as either blocking or non-blocking. Compute operations are characterized by the amount of time spent performing computation on a particular type of CPU. While this value can of course vary, we have found that a representative average is sufficient to inform *scc*; we show later in Section 6.1 that *scc* can help evaluate the sensitivity of its output to the input values. I/O operations are attributed with the dataset on which the operation is being performed and whether it is a read or write operation. Similarly, every task dependency is annotated with the invoked task.

The operations in a task's execution path may not be completely deterministic. For example, an I/O operation may hit the cache in some cases but not all, or a remote task may need to be invoked only based on the results of prior task invocations. To capture such non-determinism, every operation has an additional attribute—the probability of its execution. This, for example, enables us to capture developer knowledge of typical working set sizes for individual datasets and the hit rate on the working set.

Lastly, we also require that each task node be tagged with its memory requirements. While some applications may use all available memory and garbage collect on demand, we consider required memory to be the amount necessary to maintain performance. Note that this specifies memory that *scc* must allocate for computation beyond any additional DRAM *scc* provisions as RAM disks to store datasets.

Representing datasets. Next, we account for datasets in terms of their I/O bandwidth and capacity requirements. The I/O requirements from a dataset are determined by all the I/O operations performed on it, across the execution paths of all tasks. We ask that each I/O operation be tagged with three attributes—the number of records read or written, the number of bytes in each record, and whether records are read in parallel. The last of these three properties can be specified by the application developer, while the other two depend on the workload for which the application is being deployed. Again, we find that average values suffice for our target throughput-based SLAs. Describing I/O in terms of records accounts for the overhead seen between successive read/write operations on storage media such as disks and SSDs, e.g., from disk seeks. We similarly annotate task dependencies with three attributes—the number of invocations being performed, whether they are in parallel, and whether the whole dependency is blocking or non-blocking.

Lastly, we account for a dataset's capacity requirements by requiring that it be tagged with three additional attributes: its size, whether it must be persistent, and whether the dataset is local or remote. This last attribute differentiates between data assumed in the application's implementation to be on a storage unit local to the task accessing it as opposed to data that may be stored on a storage unit on a different machine in the cluster. Though a remote file can be made to appear local by use of systems such as NFS, we capture the application developer's

assumption of local storage, since remote access leads to higher access latencies. *scc* leverages this distinction in two ways. For a remote dataset, *scc* explicitly accounts for network load resulting from I/O requests and some CPU requirements for the machines hosting the dataset. Conversely, task-local storage constrains the amount of parallelism available on a single machine due to the storage bandwidth and number of storage unit slots available on the node.

Figure 2(c) presents an example (for another of the applications we use in our evaluation, ProductSearch, a product search Web service) of the precise format in which such an application characterization is specified as input to *scc*.

4 Implementation of *scc*

Next, we describe how *scc* processes its inputs to generate cost-effective cluster configurations.

4.1 Overview

scc determines the cost versus SLA distribution for a given application deployment by considering the configuration for each point in the distribution independently. To compute the cluster configuration for a target SLA, *scc* needs to answer two questions. First, it needs to determine the *architecture* of the cluster—for each dataset of the application, it must determine the type of media on which the dataset should be stored and how to pack the storage units into servers. This packing is constrained by the number and location of CPUs available to assign to the compute tasks that access each dataset. Second, *scc* needs to identify the *scale* at which this architecture must be instantiated to meet the SLA—scale is determined by the number of servers, storage units, and CPUs, as well as the level of parallelism of each application task.

Guiding Principles. Two key principles help *scc* identify the right cluster configuration. First, the architecture and scale for every application component can be determined independently when all operations are performed asynchronously, but not when some operations are synchronous. The SLA for any task only specifies the rate at which a task's execution path must run. In the typical case where a task's execution path contains some operations that block others, *scc* needs to determine the "division of labor" across these operations that minimizes cost. For example, in a task that reads from an input dataset and then writes to an output dataset, in order to meet the task's SLA, it may suffice to provision fast storage for any one of the two datasets; provisioning fast storage for both datasets may unnecessarily result in higher cost due to storage capacity requirements, whereas slow storage for both may incur higher costs in satisfying I/O throughput needs. Hence, *scc* jointly determines resource requirements across all application

Configuration state: $S = (S_1, \ldots, S_n)$, where
S_i = storage type assigned to i^{th} dataset
for every remote dataset d_i
compute U_i = no. of units of S_i to meet capacity and I/O needs from d_i
for every task t_i
R_i = average runtime of t_i
P_i (parallelism of task t_i) = SLA(t_i) $\times$ R_i
for every dataset d_j local to t_i,
compute no. of units of S_j to meet capacity and I/O needs from d_j for one instance of t_i
Linear integer program to choose servers
Variables:
1. booleans for whether k^{th} server is of jth type
2. $\forall$ remote dataset d_i, no. of units of S_i in k^{th} server
3. $\forall$ task t_i, no. of instances on k^{th} server
Constraints:
Per-server constraints:
1. On each I/O controller, (no. of storage units $<$ no. of slots) and (I/O throughput $<$ bus bandwidth)
2. (I/O throughput on remote datasets and local datasets accessed remotely) $<$ network bandwidth
3. no. of CPUs $<$ no. of CPU slots
Per-dataset and per-task constraints:
1. $\forall$ dataset d_i, (no. of units across all servers = U_i)
2. $\forall$ task t_i, (no. of instances across all servers = P_i)
Objective:
Minimize cost of (servers + storage units + CPUs)

Figure 4: Summary of *scc*'s procedure for determining a cost-effective cluster configuration that satisfies target SLAs, given a particular assignment of storage types to datasets.

components.

Second, since *scc* is provisioning for peak load, it prevents over-provisioning by ensuring that at least one resource is bottlenecked on every server at peak load. (If the application provider desires to run the cluster at lower peak utilization, that can be specified as input.) Based on our characterization of hardware, there are four possible bottlenecks on each server—1) the number of slots or 2) the bandwidth on an I/O controller, 3) the number of CPU cores, or 4) network bandwidth.

Algorithm. Driven by the need for joint optimization across components, *scc* represents each point in the state space of configurations by the assignment of storage unit types to datasets. As a result, if S is the number of storage choices and D is the number of datasets, *scc* has to search through a space of $O(S^D)$ configurations; for each dataset, *scc* can choose any one of the S storage options.

In cases where the configuration space is too large to perform an exhaustive search, *scc* performs a repeated gradient descent search: We start with a randomly chosen configuration. In each step, we consider all neighboring configurations—those which differ in exactly one

dataset's storage-type assignment—and move to the configuration that still meets the SLA with the maximum decrease in cost. We repeat this step until we find a configuration where all neighbors have higher cost. Since gradient descent can lead to a local minimum, we repeat this procedure multiple times with different randomly chosen initial configurations and settle on the minimum cost output across the multiple attempts. In our evaluation, we have found that repeating the gradient descent 10 times is typically sufficient to find a solution close to the global minimum. Therefore, even when determining the configuration to satisfy workloads of tens of thousands of queries per second, *scc*'s running time for any particular SLA is within a minute.

At the heart of *scc*'s search of the configuration space is a procedure—summarized in Figure 4—that, given any particular assignment of storage types to datasets, determines a cost-effective set of resources to meet the target SLAs. In this procedure, *scc* first determines for each remote dataset, i.e., not local to any task, the number of storage units required of the type assigned to the dataset in the configuration state. Second, *scc* determines the number of CPUs required by every task and the number of storage units of the assigned type needed by the task's local datasets. Finally, it determines the types of servers and number of each kind required to minimize overall cluster cost. We describe these three steps using examples from illustrative applications.

4.2 Resources for datasets

A dataset's storage resources need to satisfy two requirements: capacity and I/O throughput. To determine the cheapest storage solution that satisfies both, *scc* computes the number of storage units required to satisfy each requirement independently and chooses the maximum of the two. When the former (latter) is more expensive, we call the dataset capacity (I/O) bound. A capacity-bound dataset requires storage equal to the dataset's size irrespective of the medium used. Determining the storage required by a I/O-bound dataset is more involved. Though the total capacity of the storage units allocated to the dataset need only be equal to the dataset's size, we may need more units—under-utilizing the capacity on each of them—to meet throughput demands.

We compute I/O requirements as follows. As described in Section 3.3, the application characterization specifies the record size and the number of records read/written in every I/O operation. *scc* computes the overall number of I/O operations that a particular storage unit can support based on its rate and gap parameters. The SLA combined with the probability attributed to an I/O operation fully specifies the required frequency of the operation, which in turn determines the number of storage units required to deliver the performance in parallel.

For example, when serving requests to view photos in PhotoShare, one photo of size 200 KB on average is read from the photos dataset on every photo view. If the photos dataset were assigned to 15K-RPM disk (Table 1), which offers a read rate of 150 MBps and a read gap of 3.5 ms, it will be able to serve 200 KB-sized reads at the throughput of $\frac{200\text{KB}}{\frac{200\text{KB}}{150\text{MBps}}+3.5\text{ms}}$, approximately 40 MBps. Therefore, if the SLA specifies 1000 photo views per second, $\frac{200\text{KB}\times 1000/s}{40\text{MBps}} = 5$ units of 15K-RPM disks are required to satisfy the I/O throughput requirement.

4.2.1 Task phases

Not all tasks in an application execute concurrently, e.g., the Map and Reduce tasks run in different phases of a MapReduce job. Since datasets are subject to I/O operations only from tasks executing in a particular phase, *scc* computes the storage needed to meet I/O requirements in each phase independently. The storage requirements for a dataset during a particular execution phase are computed as the sum of storage needs across all the I/O operations made on the dataset by the tasks that run in that phase. *scc* computes the overall I/O-mandated storage requirement as the maximum over all phases.

4.2.2 Caching for higher I/O

When a dataset is I/O-bound, storing it across units of a single type may not always be the cheapest solution. I/O throughput of persistent datasets can be improved by introducing a second type of storage unit as a caching layer. For example, when considering a single storage type to service the entire load, the SSD is the most cost-effective option for the tags dataset in the PhotoShare application. However, a cheaper solution is to store the persistent copy of the tags on a 7.2K-RPM disk and to serve reads from a cached copy in DRAM.

scc assumes write-through caching. Persistent storage units handle all writes and maintain a persistent copy. Units of another type, with higher I/O rates, handle all reads. To ensure durability, every write is committed to both copies and by default, *scc* provisions enough storage to cache the entire dataset. However, developer knowledge of the application's working set size—encoded into the application specification as different capacity requirements for the dataset and for the cache—can also be used to determine what fraction of the dataset is to be cached. To evaluate whether such a solution is cost effective, *scc* computes the costs of both copies of the dataset separately and computes their sum.

4.3 Task Resources

scc next determines the resource requirements of each compute task in three steps. First, it determines the CPU utilization of the task. Second, it computes the degree of parallelism—i.e., the number of threads/processes of the

task—required to meet the SLA. Finally, it determines the number of storage units required per instance of the task for each of the task's local datasets.

A task's CPU utilization is the fraction of its run time spent performing computation. *scc* translates a task's CPU utilization into the corresponding CPU resources required by computing the level of parallelism required to meet the SLA: if a task's execution path is to be executed with frequency F and the task's average run time is R, then $(F \cdot R)$ instances of the task are required. The value of F for a task is computed from the SLA for that task and other tasks that depend on it; R is computed by appropriately summing up the times for compute, I/O, and task invocation operations in the task's execution path, taking into account, for each operation, its probability and whether it is blocking or non-blocking.

scc calculates each task's storage requirements for its local datasets based on capacity and I/O throughput requirements. *scc* also computes the task's memory requirements and the network bandwidth needed for I/O accesses to remote storage. *scc* determines each of these three requirements—local storage, memory, and network bandwidth—per instance of the task and linearly extrapolates to a target level of parallelism.

4.4 Optimizing server costs

Finally, *scc* optimizes cluster cost by minimizing the cost of required servers. Determining the servers required to host storage and CPU resources reduces to the multi-dimensional vector bin packing problem [12]. Each server type is associated with a cost and a vector of resource limits, such as the I/O bandwidth of each I/O controller and the maximum number of CPUs that the server can accommodate. Respecting these resource limits, CPUs and storage units required by tasks and datasets must be placed across servers, while minimizing total cost. *scc* solves this bin-packing problem with a linear integer program.

5 Evaluation

Next, we demonstrate that *scc* achieves the right cost versus performance tradeoff. Unfortunately, it is difficult to select appropriate comparisons. Though there exists a large body of work on capacity planning [22], all of it revolves around the question: "Given a cluster architecture for an application, how many servers of each type in the architecture are necessary?" In contrast, *scc* minimizes cost by determining not only the right scale, but also the architecture most suited for a given application deployment. Moreover, conversations with major infrastructure providers reveal that existing approaches for provisioning cluster applications used in practice are ad-hoc—the primary motivation for our work.

5.1 Methodology

We apply *scc* to three distributed applications with disparate workload characteristics to identify the cost-versus-SLA tradeoff in each case. To keep the discussion simple, we fix capacity requirements while varying the SLA. For each application, we validate the cost-effectiveness of *scc*'s output for one particular target SLA. Though *scc* readily outputs cluster configurations on the scale of tens of thousands of servers, we focus on smaller scales for validation so that we can instantiate the configurations with hardware we have on hand. Note that even at the scale of a few servers, the combination of type and quantity for storage, compute, and servers results in a very large configuration space. For example, with 5 servers of type Server1, over 10^{14} cluster configurations are feasible using the building blocks in Table 1.

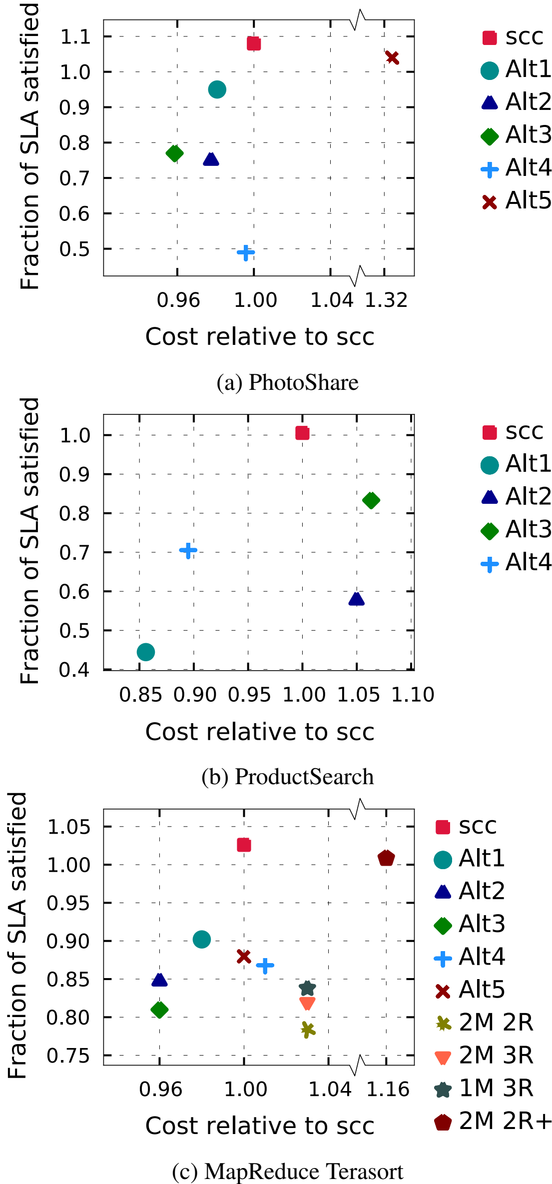
In the absence of prior approaches for principled determination of cluster architectures, our evaluation compares configurations output by *scc* with *all* possible alternative assignments of datasets to storage types; for each alternative, we consider those quantities of hardware resources to make cost comparable to *scc*. Here, we present results from alternate architectures that come closest to matching *scc* with respect to satisfaction of SLAs. In some cases, we also consider alternative architectures at the scale required to meet input SLAs and show that they incur higher costs than *scc*. For each experiment, we physically provision clusters composed of the building blocks provided as input to *scc*.

Table 1 summarizes the resources provided as input to *scc*, represented formally as in Figure 2(b). We construct our specification for cluster building blocks based on HP ProLiant DL380 G6 servers interconnected by a Gigabit Ethernet network. In each server (Server1), we consider the resource limitations to be one quad-core Intel Xeon processor, four SAS slots, and up to 12 GB of DRAM. Each of the SAS slots can support a 7.2K-RPM disk, a 15K-RPM disk, or an Intel SSD. To evaluate the performance of a given configuration, we turn off CPU cores and/or use only a subset of the SAS and DIMM slots.

For each of the resources, we consider the cost to be the amount we paid, excluding support, plus energy costs computed based on power usage numbers from product data sheets (we assume $\$0.10/kWh$ over a three year deployment). Though the power drawn by any unit can vary from its specification, we study the robustness of our results (Section 6.1) and find that they remain unchanged even if energy costs increase by a factor of two.

5.2 Photo sharing

Our first application, PhotoShare, is an interactive photo sharing application. It allows users to upload tagged photos, to view thumbnails for photos associated with a given tag, and to view the photos. PhotoShare is a



(a) PhotoShare

(b) ProductSearch

(c) MapReduce Terasort

Figure 5: Validation of cluster output by *scc* for particular SLA values in the three application cases.

C++ FastCGI application hosted on lighttpd webservers. Uploaded images are thumbnailed and stored, whereas tag updates are made via RPCs. Data is kept in a distributed log-based key-value storage system. Image, tag, and thumbnail views translate to fetches from the store. The three SLA metrics are the simultaneous rates for uploading photos, viewing photos, and viewing thumbnails associated with tags. Our input workload has, on average, 200-KB images that convert to 4-KB thumbnails, and an average of 10 photos/tag and 10 tags/photo.

We apply *scc* to study the cost as a function SLA by fixing the ratio of the rates for uploads, photo views, and tag views at 1:3:1. Figure 6 shows this cost distribution for a range of SLA values. Perhaps surprisingly, no huge spikes are observed in this distribution; this is because *scc* balances costs across the kind of storage, the number of CPUs, and the number of machines provisioned. Rather than adding more machines of the same type, the cluster architecture transitions to faster storage as the SLA becomes more stringent, with transitions in storage type for different datasets seen at different SLA values. Table 2 highlights these transitions. Note that the quantity in which different types of resources are provisioned varies within each architecture regime specified by every row in the table.

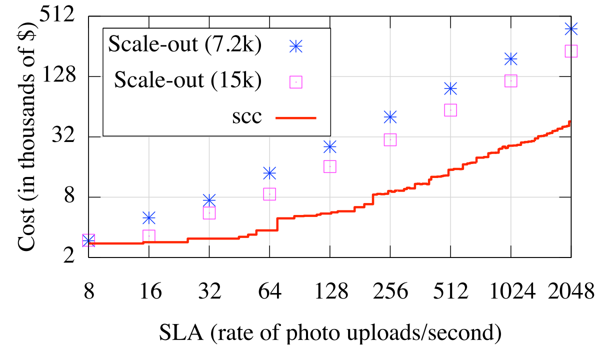


Figure 6: Cost versus SLA distribution output by *scc* for PhotoShare. Note log scale on *y* axis.

We further compare the cost output by *scc* with the cost associated with a scale-out approach. We compare the *scc* configuration to the cases where the building block is based around: 1) storage servers with four 7.2K-RPM disks (the cost-optimal storage type for all datasets at the lowest SLA), and 2) servers with four 15K-RPM disks. In either case, more storage servers are added as the required rates increase. Figure 6 shows that the costs in both cases are significantly greater than with *scc*, incurring between 3 and 4.5 times more cost (note the logarithmic *y* axis). Thus, simply scaling out a homogeneous configuration that is cost-effective at low loads can result in significant cost inflation at higher loads.

To verify the performance of *scc*'s suggested configuration, we focus on one particular SLA: 100 uploads/s, 300 photo views/s, and 100 tag views/s. The fraction of the SLA satisfied is the minimum fraction of sustained request rates across uploads, photo views, and tag views. *scc* determines the following cluster configuration for this SLA: one machine, with 4 CPU cores and 2 GB of DRAM hosts the webserver; a second machine stores the photos across four 15K-RPM disks; and a third machine hosts one SSD for thumbnails, and 1 GB of DRAM and one 7.2K-RPM disk for tags. Each of the two storage machines have 2 CPU cores and an additional 1 GB of DRAM, as required by the key-value storage system.

Figure 5(a) shows that this configuration meets

Uploads/s	Storage unit type		
	Photos	*Thumbnails*	*Tags*
≤ 5	Disk	Disk	Disk
5–25	Disk	Disk	Disk + DRAM
25–330	Disk	SSD	Disk + DRAM
330–930	SSD	Disk + DRAM	Disk + DRAM
930–10k	Disk + DRAM	Disk + DRAM	Disk + DRAM

Table 2: Different regimes based on SLA requirements in the cost-effective architecture for PhotoShare.

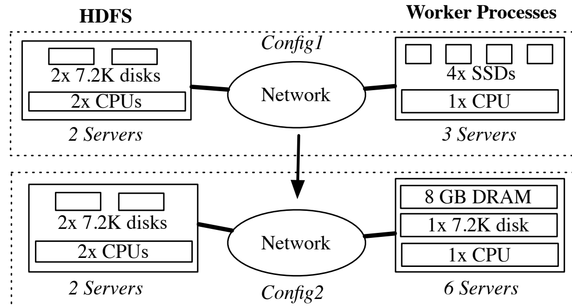


Figure 7: Transition in *scc*'s output for ProductSearch from *Config1* at 12 queries/minute to *Config2* at 13 queries/minute.

the SLA; in fact, the configuration is slightly over-provisioned. It also shows the configuration is near a minimum: removing a core from the webserver (*Alt1*), replacing the thumbnail's SSD with a cheaper 15K-RPM disk (*Alt2*), removing one of the photo disks (*Alt3*), or replacing the thumbnail's SSD with two 7.2K-RPM disks (*Alt4*) all result in SLA misses. A scale-out architecture extending *Alt4* with more 7.2K-RPM drives (*Alt5*) incurs 30%-higher cost to meet the SLA.

5.3 Product search

Our second application is a multi-merchant product search and comparison service, which we call Product-Search. We store product tables, which include product serial numbers, types, descriptions, and costs, along with product type field indices in a Hadoop Distributed File System (HDFS). In addition, user rating data is stored in a separate database table. Worker processes running across the cluster process queries for the cheapest product of a given type with a minimum user-specified rating. Each worker maintains a local copy of the ratings table as well as an index on the product serial number field; the ratings table and index are hence, specified as local datasets in the application's specification. To execute a query, a worker fetches the relevant product table and index from HDFS and then performs a join with the ratings table on the product serial number field, selecting for rows with the specified product type.

In our deployment, we build product tables with an average of 200K products, each with an average of 200 ratings. This translates to 8 GB for the ratings and roughly 800 MB for each product table. The SLA for this application specifies the required query rate.

We apply *scc* to determine system cost as a function of the SLA value. As with PhotoShare, the architecture of the cost-effective cluster changes significantly across different regimes of the SLA. At low query rates, *scc* recommends disks for both HDFS and local storage of workers. As the required query rate increases, *scc* transitions to using faster storage or provisioning more machines to handle the increased load. Figure 7 illustrates one particular transition between query rate regimes. Also, in this case as well, *scc*'s configurations yield significant cost savings compared to simple scale-out options—roughly 3× and 2× savings on average in comparison to the scaling out of homogeneous configurations with 7.2K-RPM and 15K-RPM disks, which are cost-optimal at low loads.

We validate *scc* with an SLA of 12 queries per minute. *scc*'s cluster output for this case has two parts. First, the HDFS repository is stored across two machines, each with one CPU and two 7.2K-RPM disks. Second, 12 worker processes are spread across three machines, each with one CPU and four SSDs. We run this configuration for 15 minutes. Figure 5(b), which plots the fraction of required queries completed during the experiment, shows that this configuration is able to meet the SLA.

Next, we compare *scc*'s output with alternative configurations. First, we consider clusters with alternative local storage for the workers—*Alt1* and *Alt2* use 15K-RPM drives, and *Alt3* uses 7.2K-RPM disks with DRAM. In each case, we consider the number of workers and servers to keep cost comparable to *scc*. In both *Alt1* and *Alt2*, the disk's lower random read throughput inflates query processing times and, hence, aggregate throughput falls well below the SLA. The performance of *Alt3* comes close to the SLA, but still falls short. Second, when we place all four disks underlying HDFS into one machine (*Alt4*), the 1 Gbps network becomes a bottleneck relative to the aggregate read throughput from four 7.2K-RPM drives. As a result, download times increase, leading to SLA violations.

We also use this example application to test *scc*'s ability to capture knowledge of working set sizes. We again apply *scc* to satisfy the SLA of 12 queries per minute, but this time with the additional input that 20% of product types receive 80% of queries (the application specification for this case is shown in Figure 2(c)). In this case, *scc* outputs an alternate architecture where 12 worker processes, previously run on three machines each with four SSDs, are now instead run on three machines each with four 15K-RPM disks and 10 GB of DRAM. Queries to "hot" products are served from DRAM and those to "cold" data are served from the disks. This configuration

meets the SLA with 7%-lower cost than the case where access patterns were assumed to be uniform.

5.4 Sorting binary tuples

Our final application, Terasort [29], is a MapReduce job that sorts collections of 100-byte tuples, each consisting of a 10-byte key and a 90-byte value. A *Mapper* reads tuples from a local input file and sends them over the network to appropriate *Shuffle* processes. Each Shuffler writes the tuples it receives to a set of intermediate, sorted local files. Once the Mappers and Shufflers are done, the Shuffle processes transform into the role of *Reducers*. Each Reducer merges the tuples in the local files into an output file of sorted tuples. For this application, the SLA is the total runtime of the MapReduce job.

We use *scc* to determine the cost of clusters capable of sorting 50 GB for a range of runtimes. Note that though we put together clusters of individual servers here, we envision that *scc* will be used for such jobs to provision a set of virtual machines in a virtualized infrastructure. Unlike PhotoShare and ProductSearch, we see no significant architecture changes over different runtimes. *scc* uses the basic building block of provisioning Mappers on machines with four cores and one 7.2K-RPM disk and Shufflers/Reducers on machines with four cores and two 7.2K-RPM disks. *scc* provisions more machines for both components to meet more stringent SLAs. Faster storage has no benefits because the job is CPU bound.

Next, we verify the performance of the cluster output by *scc* for an SLA that requires 50 GB to be sorted in 25 minutes—an average sorting rate of 2 GB per minute. The *scc* cluster consists of 8 Mappers and 16 Reducers spread across two and four machines respectively with the above-mentioned building blocks. We run the application on this cluster to sort 50 GB of input data. Figure 5(c) plots the SLA-specified runtime divided by the observed runtime and shows that the *scc* cluster meets the SLA.

To evaluate the cost-effectiveness of *scc*'s output, we also sort 50 GB of data on several alternative architectures. A few such alternatives include *Alt1* and *Alt2*, which reduce the number of cores from 4 to 3 on the Mapper machines and on the Reducer machines, respectively. *Alt3* substitutes the two 7.2K-RPM disks on each of the four Reducer machines with one 15K-RPM disk shared between the intermediate and output data. Figure 5(c) shows that the runtime of the Terasort job misses the SLA by at least 10% in every case. The figure also shows that two other alternatives—*Alt4* and *Alt5*—which have similar cost to *scc*'s output but trade off compute resources for more or faster storage, also fall short.

Unlike our other two example applications, compute-intensive MapReduce jobs have a cluster configuration recommended by best practices. We modify the cluster

Attribute	**Range with same architecture**		
	Lowest value	**Input value**	**Highest value**
Avg. photo size	50 KB	200 KB	850 KB
Avg. thumbnail size	1 KB	4 KB	30 KB
SSD unit price	$200	$450	$900

(a)

Dataset	**Most sensitive to what change in hardware costs?**
Photos	20% drop in $ of 7.2K-RPM disk
Thumbnails	92% drop in $ of DRAM
Tags	31% drop in $ of 15K-RPM disk

(b)

Table 3: Determining robustness of *scc*'s output with respect to its input: (a) robustness of cluster configuration with respect to input values for a sample set of attributes, and (b) the change in hardware costs to which *scc*'s storage decision for each dataset is most sensitive.

architecture to be six machines each with four cores and two 7.2K-RPM disks—a setup recommended by Cloudera for a "Balanced Compute Configuration" [13]. Also, we configure every node in the cluster to run a fixed number of Mappers and Reducers. We evaluate three different combinations of Mappers and Reducers per node (the *"2M 2R"*, *"2M 3R"*, and *"1M 3R"* points in Figure 5(c)), and interestingly, we find that the recommended MapReduce configurations deliver lower performance than *scc* for similarly priced clusters. While all three alternatives meet the SLA when scaled out to an additional machine, e.g., the *"2M 2R+"* point in the figure, this results in 16%-higher cost than *scc*'s recommendation.

6 Discussion

In this section, we discuss the robustness of *scc*'s output, its utility in planning application implementation architectures, and its extensibility on other fronts.

6.1 Robustness of *scc*'s output

scc's output cluster configuration for a target SLA is a function of both the SLA and the *exact* values specified for the various attributes in the application and hardware specifications. In practice, a user of *scc* may not have precise values for all attributes due to incomplete knowledge of the application workload, uncertainty of hardware costs, or measurement inaccuracy in benchmarking.

scc is naturally built to cope with such uncertainty. For every attribute in the input specifications, *scc* varies the value of the attribute in the neighborhood of the initially specified value. For each attribute, it then outputs the range of values for that attribute wherein the cost-effective cluster architecture, i.e., the types of resources assigned to different application components, remains

unchanged; variance of the attribute's value within this range can be handled by simply adding more resources of the same type. Outside of that range, the cluster will need to be revamped with a different type of resource for some application component, a significantly more cumbersome undertaking. For example, we again consider PhotoShare with an SLA of 100 uploads/s, 300 photo views/s, and 100 tag views/s. Table 3(a) shows the value ranges output by *scc* for a few attributes, within which the cluster architecture is robust to change. For example, we see that as long as average photo size remains between 50 KB and 850 KB, the cluster architecture remains the same as that obtained with the input value of 200KB.

Furthermore, *scc* can also evaluate the sensitivity of its choice of storage configuration for every dataset in the application. For example, consider PhotoShare again with the same input SLA as above. Based on current hardware costs, *scc* determines that photos be stored on 15K-RPM disks, thumbnails be stored on SSDs, and tags be stored persistently on 7.2K-RPM disks and cached in DRAM, in order to meet the SLA at minimum cost. However, these recommendations are likely to change as prices for storage units drop. *scc* can determine how robust are its choice of storage options to such changes in hardware prices. To do so, it varies the price of every type of storage unit from its input value down to 0, and notes the inflection points at which the optimal storage choice for some dataset changes. Based on this analysis, it can determine, for every dataset, that change in hardware price to which the current storage choice for the dataset is most sensitive. Table 3(b) shows the output of this analysis for the three datasets in PhotoShare. While the storage choices for photos and tags are sensitive to relatively small reductions in the prices for 7.2K-RPM and 15k-RPM disks, *scc*'s recommendation of storing thumbnails on SSDs is very robust to price fluctuations.

6.2 Informing application development

Thus far, we assumed a fixed application implementation. However, *scc* can also help determine the best application architecture. For instance, in the case of Terasort, there is a fundamental performance tradeoff between a cluster configuration with sufficient DRAM to store all data to be sorted and one that must stage portions of the data into memory from secondary storage. The former case requires one read and one write of all the data while the latter requires two reads and two writes of the data [3].

To explore cost–performance tradeoffs for the two application architectures, we must consider the benefits of servers with more network bandwidth (so remote storage does not become a bottleneck) and more memory (to allow for storing the entire dataset in memory). In Table 1, Server2 is the same HP ProLiant DL380 G6 server as Server1, but with more resources per server and a 10-Gigabit Ethernet (10GigE) NIC. Server3 is the HP ProLiant DL785 G5 Server, which accommodates more processors and DRAM, again with a 10GigE NIC.

We use *scc* to determine the cluster configuration necessary to sort 100 TB in the time required to read/write the whole data from/to disks twice at the read/write rate of the 7.2K-RPM disk. This cluster costs $239K and completes sort in 10,000 seconds. For the alternative implementation where all data fits in DRAM, we apply *scc* to satisfy the SLA of sorting the complete dataset in half the SLA of the baseline implementation. The cheapest cluster configuration determined in this case costs $5.6M and sorts 100 TB in 5,000 seconds. Thus, according to *scc*, the latter implementation provides a $2\times$ speedup at $24\times$ the cost. The application designer can decide if the faster processing is worth it.

6.3 Extensibility of *scc*

Our approach of determining cost-effective cluster configurations with *scc* is extensible in several ways.

Less flexible infrastructure services. Though we restrict our attention in this paper to flexible infrastructure services that permit arbitrary mixing and matching of compute and storage resources on a per-server or per-VM basis, *scc* can also be readily applied to less flexible services that offer only certain combinations of processor, storage, and memory configurations, e.g., Amazon's EC2 service [1]. In such cases, each combination of resources offered by the infrastructure service can be provided as input to *scc* as a separate server type, and the cost of each server will subsume the costs of all the resources that come with it.

Accounting for availability. Though we have focused on performance requirements of applications thus far, performance and availability SLAs need to be considered in unison. For example, a cheap disk type may be an attractive option for a capacity-bound dataset but the degree of replication necessary to meet availability goals may make the option cost-prohibitive. *scc* can be extended to pick for each dataset that combination of storage type and associated replication factor that meets the combination of performance, availability, and consistency requirements at minimum cost.

Load variation and incremental growth. Our current implementation of *scc* provisions applications for peak load. However, when the distribution of load across time is available, *scc* can leverage the information in two ways. First, *scc* can estimate energy costs more accurately. Second, when pricing for resources is "elastic", i.e., a user can provision resources on-demand and pay for what she uses, *scc* can make incremental reconfiguration decisions, determining when to simply scale-out

and when to switch between architectures. *scc*'s distinction between remote persistent datasets and local transient datasets enables it to capture the costs associated with data redistribution.

Network configuration and CPU diversity. *scc*'s specification of application behavior can be used to infer the communication pattern among the application's components, and thus inform configuration of the cluster's network. For example, in the case of ProductSearch, *scc* can infer from the application specification that the workers communicate only with the HDFS repository but not among themselves. *scc* can then use this information to recommend a bi-partite network with servers hosting HDFS on one side and servers hosting workers on the other side. *scc* can also be readily extended to choose among a range of CPUs; the application specification simply needs to include for every compute operation the time required for that operation on each type of CPU.

7 Related work

Our work builds upon and shares some similarities with several lines of prior work.

Tuning storage: Minerva [5], Hippodrome [7], and Rome [34] automate the provisioning of disk arrays with a similar approach of characterizing workloads and storage. Ursa Minor [2] varies erasure coding parameters depending on an application's availability requirements. PADS [9] is configurable to build a wide range of replication systems with varying consistency semantics. In contrast to all of these efforts, we consider an application's storage and compute requirements in unison. Moreover, we choose among different storage media such as disk, SSD, and DRAM to minimize cost, with multiple media possibly being used for the same application.

Application modeling: Bodik et al. [10] infer application performance models by applying machine learning techniques on statistics gathered by monitoring the application execution. Thereska et al. [31] predict performance across application configurations based on statistical models. IRONModel [32] corrects deviations between the performance of running systems and high fidelity models. In all cases, since application models are tuned to specific cluster configurations, they are not directly applicable to alternative hardware configurations.

Stewart and Shen [30] build performance models of multi-component applications to aid in the placement of application components on a given cluster. Osogami and Itoko [25] apply hill-climbing techniques to automatically determine web-server parameters, and Liu et al. [20] construct a queuing model for a three-tiered web service to predict throughput and response times. Again, all of these consider a fixed hardware configuration.

Application-specific cluster architectures: Application developers have converged on a range of cluster architectures for individual applications. Several web services employ DRAM caches using distributed in-memory storage systems [21, 26]. Applications such as WER [16] use clusters that have separate sets of machines for compute and storage. FAWN [6] and Gordon [11] use SSDs to build performant yet power-efficient distributed data processing systems. MR-Perf [33] and Starfish [18] use an approach similar to *scc* but focus solely on predicting cluster requirements of MapReduce setups. *scc* not only infers these cost-effective architectures for existing applications, but also enables the inference of the right cluster architecture for emerging applications.

Storage and computing services. There been a few recent attempts [19, 15] at satisfying SLAs in the setting of a compute and storage cluster shared across applications. Such multi-application environments have also seen the recent emergence of virtual storage appliances. *scc* is targeted at the still significantly more common scenario of cluster deployments for a single application.

8 Conclusions

The thesis of our work is that deployment of applications on clusters is more cost-effective if informed by characterizations of application behavior and hardware properties. Towards this end, we presented how these inputs can be specified, and we developed *scc* to compile these inputs into cost-effective cluster configurations. Our experiments in applying *scc* to a range of application workloads and storage options show that *scc* captures sufficient detail to prescribe the right combination of storage and server hardware at the right scale; modifying the architecture or reducing the scale leads to significant performance degradation. To meet application demands, *scc* often predicts heterogeneous cluster architectures that result in significant cost savings compared to simply scaling out homogeneous architectures. We plan to apply *scc* to other popular applications to determine more fine-grained characteristics from which it could benefit, and use *scc*'s application specification to select appropriate CPUs and optimize network costs. We also plan to develop tools to make it easier to put together hardware and application specifications.

Acknowledgments

We thank the anonymous reviewers and our shepherd Lakshmi Bairavasundaram for their feedback on previous versions of this paper. We also thank Brian Cooper, Khaled Elmeleegy, Garth Goodson, and Kaladhar Voruganti for their input at various stages of this project. This research was partially supported by a NetApp Faculty Fellowship.

References

[1] Amazon EC2 instance types. http://aws.amazon.com/ec2/instance-types.

[2] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: Versatile cluster-based storage. In *FAST*, 2005.

[3] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 1988.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity, data center network architecture. In *SIGCOMM*, 2008.

[5] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. A. Becker-Szendy, R. A. Golding, A. Merchant, M. Spasojevic, A. C. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 2001.

[6] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, 2009.

[7] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. C. Veitch. Hippodrome: Running circles around storage administration. In *FAST*, 2002.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, 2004.

[9] N. Belaramani, J. Zheng, A. Nayate, R. Soul, M. Dahlin, and R. Grimm. PADS: A policy architecture for data replication systems. In *NSDI*, 2009.

[10] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *Proc. of the 1st workshop on Automated control for datacenters and clouds*, 2009.

[11] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS*, 2009.

[12] C. Chekuri and S. Khanna. On multi-dimensional packing problems. *SIAM Journal on Computing*, 2004.

[13] Cloudera. Cloudera´s support team shares some basic hardware recommendations. http://www.cloudera.com/blog/2010/03/.

[14] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[16] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *SOSP*, 2009.

[17] Google Merchant Center. http://www.google.com/merchants.

[18] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. In *VLDB*, 2011.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[20] X. Liu, J. Heo, and L. Sha. Modeling 3-tiered web applications. In *MASCOTS*, 2005.

[21] Memcached. http://memcached.org.

[22] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Capacity planning and performance modeling: from mainframes to client-server systems*. Prentice-Hall, Inc., 1994.

[23] NetApp Inc. Microsoft Exchange 2007 deployment for 2,000 to 5,000 users integrated with high availability, backups, and disaster recovery. http://media.netapp.com/documents/ra-0001-0509.pdf.

[24] NetApp Inc. Oracle database dev/test reference architecture using data guard and SnapManager for Oracle deployment guide. http://media.netapp.com/documents/ra-0002.pdf.

[25] T. Osogami and T. Itoko. Finding probably better system configurations quickly. In *SIGMETRICS*, 2006.

[26] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *SIGOPS OSR*, 2009.

[27] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST*, 2008.

[28] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 1992.

[29] Sort benchmark home page. `http://sortbenchmark.org/`.

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

[31] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. In *SIGMETRICS*, 2010.

[32] E. Thereska and G. R. Ganger. IRONModel: Robust performance models in the wild. In *SIGMETRICS*, 2008.

[33] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *MASCOTS*, 2009.

[34] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *IWQoS*, 2001.