System-Level Characterization of Datacenter Applications

Manu Awasthi, Tameesh Suri, Zvika Guz,
Anahita Shayesteh, Mrinmoy Ghosh, Vijay Balakrishnan
Samsung Semiconductor, Inc.
601 McCarthy Blvd., Milpitas, CA
{manu.awasthi, tameesh.suri, zvika.guz, anahita.sh, mrinmoy.g, vijay.bala}@ssi.samsung.com

ABSTRACT

In recent years, a number of benchmark suites have been created for the "Big Data" domain, and a number of such applications fit the client-server paradigm. A large volume of recent literature in characterizing "Big Data" applications have largely focused on two extremes of the characterization spectrum. On one hand, multiple studies have focused on client-side performance. These involve fine-tuning server-side parameters for an application to get the best client-side performance. On the other extreme, characterization focuses on picking one set of client-side parameters and then reporting the server microarchitectural statistics under those assumptions. While the two ends of the spectrum present interesting results, this paper argues that they are not enough, and in some cases, undesirable, to drive system-wide architectural decisions in datacenter design.

This paper shows that for the purposes of designing an efficient datacenter, detailed microarchitectural characterization of "Big Data" applications is an overkill. It identifies four main system-level macro-architectural features and shows that these features are more representative of an application's system level behavior. To this end, a number of datacenter applications from a variety of benchmark suites are evaluated and classified into these previously identified macro-architectural features. Based on this analysis, the paper further shows that each application class will benefit from a very different server configuration leading to a highly efficient, cost-effective datacenter.

Categories and Subject Descriptors

C [**Computer Systems Organization**]: Performance of Systems; C.4 [**Performance of Systems**]: [Design studies; Performance attributes; Measurement techniques]

Keywords

Performance measurement, Datacenter Performance, Datacenter, Benchmarking, Workload Characterization

1. INTRODUCTION

In recent years, a large amount of the world's compute and storage has been pushed onto back end datacenters. The large scale of these datacenters come at a hefty price– initial setup cost can range anywhere from upwards of US $200 million, and yearly operation cost is on the order of millions [30]. With such a high TCO (Total Cost of Ownership), performance/$ is a first-order design constraint. Tailoring the hardware to the specific set of applications the datacenter is expected to run can save millions of dollars.

The term "Big Data" refers to the explosion in the quantity (and sometimes, quality) of available and potentially relevant data, largely because of the result of recent advancements in data recording and storage [26]. Modern datacenters execute a diverse set of applications on these massive datasets. These applications are collectively referred to as "Big Data" applications. Due to the unprecedented scale of these applications and their highly-distributed nature, the characteristics of "Big Data" applications are significantly different than those of traditional multi-core applications [41]. To gain better understanding of the behavior of these applications, a large body of work has been done in recent years to develop and characterize representative benchmarks for the "Big Data" domain.

A lot of these benchmark suites are focused towards research of the microarchitecture of a single server [34, 27, 41]. While they provide valuable insights into core and chip design, they usually focus on the core [1] microarchitecture analysis. Since most on the microarchitecture research is done using simulators, these "Big Data" benchmarks have been tailored to fit this research model. For example, this leads to decreasing the size of the working set so that the benchmarks can be run within a simulator. This, in turn makes the benchmark not exhibit most of its inherent properties due to which it was chosen as a representative application in the first place.

On the other side of the spectrum, detailed analysis has been published on the client performance of specific applications [13, 7, 6, 9], focusing on tailoring the application for a predefined server architecture. The middle of the spectrum– namely characterizing the behavior of full-scale applications, with representative data sets in a multi-server environment for the purposes of system design is left somewhat unexplored. In this paper, we try to provide a framework for the datacenter designer to make judicious decisions about hardware acquisition based on characterization of realistic

---

[1]This paper uses core and processor interchangeably in the rest of the discussion.

applications. This paper describes the authors' experience in compiling and characterizing a number of representative "Big Data" applications, from different benchmark suites. The authors found that the existing benchmarks, if used *as-is*, are ill-fitted for system and storage architecture research. This is either because they are scaled down to the point where they do not stress the relevant components, or because they are concentrating on studying processor microarchitecture, which is just one of the components of research on system design. The reader is taken through the process the authors went through for tuning the workloads to the environment they are running on, describing how they reason about application scaling, and presenting performance characterization and analysis for several multi-server, real-world applications.

While the results and tuning process presented here should be of interest to any hardware system researcher looking at the "Big Data" domain, they are also valuable outside the realm of pure systems research. Indeed, when trying to decide on specific hardware needed for a new datacenter deployment, server microarchitecture optimizations are of little value. Instead, this paper illustrates how a smaller number of coarse-grained metrics allow for good classification of a workload and are usually enough to make the first order decision about the most important system level bottlenecks for the application. Keeping this context in mind, this paper makes the following contributions:

- The reader is taken through the tuning process of "Big Data" applications, providing an ordered recipe for what should be done, and how.
- Characterization results are presented for several "Big Data" workloads, concentrating on the coarse-grain, per-server, system-level behavior rather than the fine-grained microarchitectural profile.
- The paper illustrates how a few coarse-grained metrics (macroarchitectural properties) are enough to gain understanding of an application's behavior, and explain how these metrics can be obtained.
- Finally, some insights are presented on how to use information about the macroarchitectural properties of applications to intelligently co-locate applications.

The rest of this paper is organized as follows: Section 2 presents an overview of the organization of a modern datacenter, and describes the applications considered in this paper. Section 3 identifies the four macroarchitectural parameters, and provides insights into why these parameters are important. Next, Section 4 presents the characterization methodology and authors' experience on performance tuning of different workloads. Section 5 presents the related work, and Section 6 concludes.

2. DATACENTERS AND APPLICATIONS

Datacenters are broadly classified into three categories: (1) Enterprise, (2) Cloud Computing, and (3) Web 2.0. While they share a common goal of reducing TCO and increasing efficiency of available resources, their application set and requirements are significantly different. As a result, the characteristics and design points of these three deployments are also very different.

- *Enterprise*: These datacenters support corporate and financial environments for primarily one institution. Such datacenters have a comparatively smaller scales, with low multi-tenancy and a smaller application diversity. Enterprise datacenters mostly run proprietary applications like SAP ERP, Microsoft Sharepoint and Microsoft Exchange.
- *Cloud Computing*: Cloud computing datacenters provide a virtualized computing and storage resources as a service to end users. These datacenters have larger scale and support a bigger class of applications. Examples of Cloud Computing services include Amazon Web Services, Microsoft Azure and Google Compute Engine.
- *Web 2.0*: Web 2.0 datacenters support massively high-volume data and users through vertically integrated application stacks and popular open-source frameworks. Most of the services provided by the likes of Google, Facebook and Yahoo! are hosted on such servers.

This paper focuses on the *Web 2.0* category– a category with an expected Compound Annual Growth Rate (CAGR) of 17.8% [24]. These datacenters typically consist of several types of servers (or server clusters), each of which are designed to execute designated tasks [39]. Figure 1 shows a representative deployment scenario, where the datacenter back-end consists of three distinct layers. Each layer runs a different class of applications and has its distinct set of hardware requirements. The following sections describe each one of these three layers, and describe the applications that have been characterized and analyzed in this paper.

2.1 The Caching Layer

The caching layer is primarily responsible for the performance and response times of the datacenter. The caching layer resides in between the web front-end and the back-end database, and uses large amounts of DRAM capacity to cache frequently used data and reduce the amount of queries hitting the back-end database servers. The reduction in database (DB) queries reduces the pressure on the servers– notably the I/O subsystem– and considerably improves user-experience. Memcache [28]– an open source key-value distributed caching software, is by far the most deployed web caching application in this domain. Section 4.2 presents the authors' experience in tuning Memcache and using it as a representative web caching layer application.

2.2 The Analytics Layer

The analytics layer contains a rich set of applications that analyze huge amount of data (i.e. "Big Data") to extract knowledge and provide value. The majority of "Big Data" applications belong in this layer. These applications can be further classified into two separate groups: *real-time* and *offline*.

Real-Time Analytics: Designed for the most latency sensitive analytics, real-time analytics applications are designed such that they store either the *entire* data set or the *vast majority* of their data set in server memory. Hence, this paradigm of Real-Time Analytics is often referred to as In-Memory Computing (IMC). In the past, their adoption was hampered by high memory acquisition cost, but with the recent decline in DRAM prices and the availability of mature software, IMC is becoming widely used in many datacenter deployments. Indeed, the exponential growth in data and the need for faster turnaround in extracting meaningful
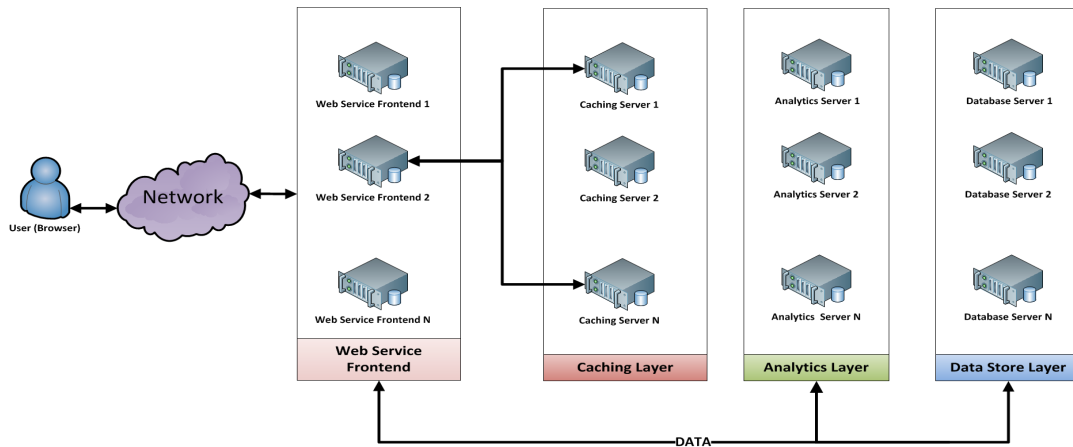
Figure 1: Representative datacenter architecture. A datacenter is composed of several different server tiers.

information is expected to expand adoption of IMC, forecasted to reach over \$13 B by 2018 [18]. Redis [4] and Apache SPARK [15] are two popular open-source, in-memory frameworks. This paper focuses on the Redis framework, which is used in the production stack of several *web 2.0* companies like Twitter, Craigslist, Flickr and Github [21].

Redis [4] is an open-source, in-memory data structure framework that provides an advanced key-value abstraction. Unlike traditional key-value systems where keys are of a simple data type, Redis supports complex data types: keys can contain hashes, lists, sets, and sorted sets. It enables complex atomic operations on these data types; all operations are executed on the server, and the entire dataset is served from the server's DRAM. Data persistence is achieved through frequent syncs to hard-drives, while sharding is used for scaling-out. This abstraction has proved particularly useful for multiple latency-sensitive tasks.

Offline Analytics and Batch Processing: Batch processing of large scale data via distributed computing is a widely used service in contemporary datacenters. These services typically rely on the Map-Reduce computational model, for which Apache Hadoop [1] is the most popular open-source implementation. A Hadoop deployment spans multiple nodes and comprises of multiple modules, including the Hadoop distributed file system (HDFS) for large-scale data storage and the map-reduce programming model for computation. While the computation is slower than an IMC system, the distributed model allows it to process massive amount of data using commodity hardware. It is extensively used by organizations to support analytics like sentiment analysis and recommendation engines on enormous data volumes. This paper studies two popular applications on the Hadoop framework:

- **Nutch** Web search [14, 32] was the first proof-of-concept Hadoop application and remains a good, representative data-intensive distributed application. Nutch is an extensible and scalable open-source web search engine package that aims to index the World Wide Web as effectively as commercial search services.
- **Mahout Data Analytics** is a scalable, open-source machine-learning framework built on top of Hadoop Map-Reduce. It is used to apply machine learning models to massive volumes of data that do not fit the

traditional computational environments. Specifically, our workload uses the bayes classifier algorithm to execute text analysis and clustering on large datasets.

2.3 The Data Store Layer

This layer often comprises of multiple databases that store and serve data back to various applications in the datacenter. Traditionally, relational databases, organized in a row-column format, were used exclusively for data storing. However, since relational databases are both hard to scale out and unfit for unstructured data, contemporary datacenters rely on NoSQL databases to meet the performance requirements at scale. This paper characterizes Cassandra– a popular NoSQL database. Cassandra is designed for write-intensive workloads and targets linear throughput scaling with the number of nodes. The YCSB [25] framework was used as a client for all experiments.

3. MACRO ARCHITECTURAL PARAMETERS FOR SYSTEM WIDE ANALYSIS

This section attempts to provide a framework for system wide analysis of representative datacenter applications. Creation of such a framework is an important step in characterizing these applications with the aim of finding the best hardware configuration for each tier and/or application class. Going through this exercise would be essential for the following scenarios: (i) a new datacenter has to be set up, based on the requirements of the application set of each tier, or (ii) a datacenter installation already exists and the applications running at each tier are well known, but the hardware infrastructure of a particular (or all) tier(s) has to be updated.

A large body of work has been dedicated to characterizing datacenter level applications. Section 5 presents a detailed survey of prior work. However, most of the existing literature is of little use for the said purpose since it highlights one of the two characteristics: (i) client side performance, or (ii) detailed exploration of server microarchitecture. Both of these levels of characterization have value, but looking at application characteristics from a system perspective is much more valuable to the datacenter designer. Getting a complete picture of how a server should be provisioned based on its behavior under real-world use cases is essen-

tial to make sure that each server machine can handle all requests, meet QoS and SLA requirements, while still not being *extremely* over-provisioned. In order to make an informed decision, according to the authors' experience, a first level analysis of datacenter applications should be done on a coarser, system-wide level. This section describes the four main macro-architectural resource categories that should be considered while making per-server provision decisions for a particular application/class of applications.

Each macro-architectural resource category is composed of several fine-grained metrics that effectively decide the nature of the application considered with respect to that category. Classifying each application into one or more such categories enables intelligent and cost-effective hardware provisioning. Moreover, since it is in-feasible to list *all* the fine-grained parameters, applications are broadly classified based on their *intensity* of usage of each resource category. The intensity of usage is defined as follows for each resource category.

CPU Intensity CPU utilization provides a broad measure of the amount of work done by the application. It is very important to note *how* the CPU is being utilized. It is entirely possible that some programs have high CPU utilization, while making little forward progress, e.g. in cases where multiple threads of a program are waiting for I/O. Learnings from the application compute requirements provides an effective understanding of the CPU performance requirements and supports informed provisioning decisions about choice of CPUs.

DRAM Intensity At a macro level, this is comprised of two main components: (i) capacity, and (ii) bandwidth. Identifying the dominating component for each application supports effective provisioning trade-offs about DRAM capacity and performance in the server node.

Disk I/O Intensity The two most fundamental components of Disk I/O are (i) access latency, and (ii) bandwidth. Understanding application throughput and sensitivity to each parameter is essential in selecting the correct interface and media for storage provisioning. These may include traditional hard-disk drives (HDDs) or flash-based solid-state disks (SSDs) on standard SATA3 interface, high performance PCIe based SSDs or a combination of some/all of the above.

Network Intensity All nodes in a datacenter communicate over a network, and ethernet is the most commonly deployed technology. Maximum bandwidth of each ethernet port is predefined by the protocol and understanding of sustained and peak application network requirements supports better network provisioning.

Several applications used in production environments representing each tier were evaluated and classified into the four broad categories. Results of characterization experiments suggest that each of the aforementioned tiers of a datacenter have distinct hardware requirements and a uniform, general-purpose hardware allocation across all tiers does not represent an efficient datacenter design. The authors provide a performance architect's perspective of the Web 2.0 applications that over-provisioning hardware in isolation to application requirements does not have any impact on the performance or scalability of the applications, and macro-architectural classification of applications is instrumental in designing and deploying cost efficient datacenters.

4. DATACENTER APPLICATION CHARACTERIZATION

Section 2 postulated that in most cases, matching the application to the right set of hardware resources requires categorizing it into one of four main categories. This section first presents the authors' experiences in installing and running out-of-the-box versions of "Big Data" benchmarks, with the default data sets. Next, the section presents the observations from the default experiments and then compares those results to finely tuned versions of the same applications to highlight the learnings from application tuning. This experience is used to show that stock versions of the considered benchmarks do not stress the *intended* subsystems of a typical server hardware. Based on these experiences, it is also shown that in a datacenter setting, one application can have multiple points of saturation on a server, and the points of saturation are highly dependent on the considered use case.

4.1 Methodology

The configuration used for these studies comprises of a cluster of eight server class machines running Ubuntu 12.04.5. Each server node consists of a dual-socket Intel Xeon E5-2690 processor. There are 8 physical cores on each socket, supporting a total of 32 independent threads per node with hyper-threading. Details of the setup can be found in table 1. The Hadoop setup consisted of seven worker nodes (DataNodes, TaskTrackers) and one master node (NameNode, SecondaryNameNode, JobTracker). The nodes are connected through a 10GbE network. The rest of the workloads use the specified number of nodes as indicated for each configuration. Most client-server style workloads were run as a single client, single server setup, unless otherwise specified.

Table 1: Server node configuration

Processor	Xeon E5-2690, 2.9GHz, dual socket-8 cores
Storage	3× SATA 7200RPM HDDs
Memory Capacity	128 GB ECC DDR3 R-DIMMs
Memory B/W	102.4GB/s (8 channels,DDR3-1600)
Network	10 Gigabit Ethernet NIC
Operating system	Ubuntu 12.04.5
Hadoop version	1.2.1
Memcache version	1.4.13
Cassandra version	2.0.9
Redis version	2.8.12

Benchmarks are evaluated by collecting data from the application, operating system and processor performance counters. High level statistics of application and cluster performance are reported as part of application client or load generator. In addition, collectl [16] is run to monitor system level performance and use its plotting utility colplot [17] to generate graphs.

4.2 Application Tuning

Most of the applications described in prior sections required a significant effort to identify and tune several parameters to emulate real-world use cases. This section summarizes the authors' learnings from the tuning process.

For most cases, if the application fit the client-server paradigm, the goal was to maximize per-client performance, usually measured through metrics like transactions/second (TPS).

Table 2: Key parameters that had to be tuned for Memcache and Cassandra

| Parameter	Original Value	Tuned Value
Memcache Tuning		
DRAM Size	4 GB	60 GB
Server Threads	4	32
Server TCP Connections	200	4096
Memcslap Clients	1	4
Threads/Client	4	16
Cassandra Tuning		
Commit Log	Same disk as DB	Different disk from DB
Heap Size/JVM Size	Default	600MB/6GB
Concurrent Reads/Writes	Default	64/64
Memtable size	Default	4 GB
Allowed open file handles	Default	32768
YCSB client threads	Default	200
YCSB Database size	Default	128 GB

The server's characteristics are reported after the application tuning has been completed.
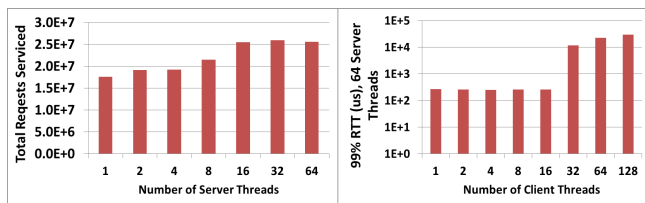
Figure 2: Memcache performance tuning for number of client and server threads.

Memcache The Data Caching benchmark provided as a part of CloudSuite benchmark suite was set up using the default parameters suggested in the CloudSuite literature. Over time, the authors realized that to emulate real-world use case, a number of parameters in the benchmark required significant changes. First, CloudSuite instructions [11] prescribe running Memcache with 4 GB memory which can be set using the `-m` option, and then increasing it to 10 GB for larger working sets. However, experiments on Xeon servers indicated that running Memcache with as much DRAM as possible leads to better client side throughput. This is because of two factors: (i) Larger DRAM capacity implies a larger cache, leading to higher hit rates, and (ii) having more DRAM at hand allows the application to do better memory management by itself. Hence, >90% of the total available DRAM was attached to the Memcache server.

Prior studies [42] indicate poor scalability of Memcache with increase in the number of server threads. However, with the current production version of Memcache, good scalability was observed in client side performance with increasing the number of threads. Client side performance does not begin to degrade until there were enough hardware contexts to run client side threads. As indicated in Figure 2, it was possible to maintain 99th percentile latency of < 1 ms with 32 server threads. This coincides with the E5-2690 Xeon server having 16 physical and 32 SMT threads The number of TCP connections also had to be tuned for both the server and the client. The optimization details are listed in Table 2. This experience led to an exploration of the optimal number of threads on both the server and the client.
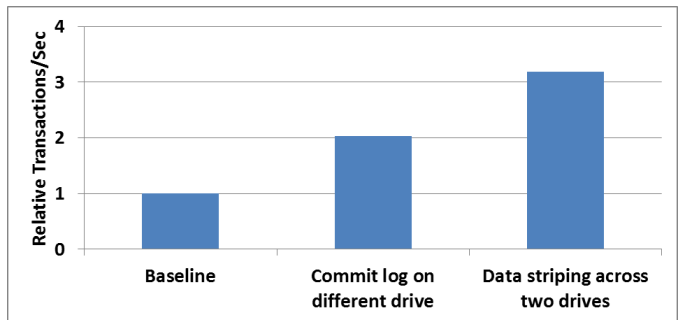
Figure 3: Optimized Cassandra client side performance for YCSB workload E; parameters in Table 2.

In order to reap maximum benefits out of a datacenter's Memcache installation, the administrators should ensure the upper limits of TCP connections that both servers and clients are allowed to make. One of the most important changes that the authors had to do to their setup was increase the number of client machines. This was because of two reasons: (i) one physical client was not enough to push the limits of the server, and (ii) no realistic use case could be achieved using a one-server, one-client setup, especially for Memcache [23]. Hence, the experiments were run with four physical clients.

Cassandra Cassandra needs tuning in two different ecosystems: (i) Being a client-server database application, the application server needs to be tuned to make sure that the destructive interference between different components of the application is reduced and the server is given enough physical resources for it to function properly – this will be explained shortly, and (ii) since it is a Java application, the JVM settings need to be tuned. The latter is relatively easy since enough information is publicly available.

For Cassandra-specific configuration parameters, tuning three main ones in accordance with available resources was found to have most impact on performance. The first and the most important optimization was to have the commit log on a different disk than the actual database. Since Cassandra is optimized for extremely fast writes, which are considered complete when they are written (simultaneously) to an on-disk structure called the commit-log (for durability) and also to an in-memory structure called the *Memtable*, which is a write-back cache. The second related optimization is the size of the Memtable. A small Memtable causes frequent writes to disk, affecting overall performance. The authors found that for their setup, 4GB was the most optimal Memtable size, and any further increase led to diminishing returns. Combined, these two optimizations lead to a significant gain in performance (2×, as indicated in Figure 3; left and center bars). This is because the destructive interference between the the two types of disk I/O : one for durability to the commit log and the other to propagate the writes to the database is removed. In addition, striping the database across more than one disk also leads to much better performance – middle and right bars in Figure 3 show that another 1.5× performance benefit can be realized by just allowing Cassandra to stripe data across two HDDs.

Several other, albeit smaller optimizations were also needed for correctness, rather than performance. For example, the

maximum number of concurrent open files was increased within Linux to 32768, otherwise Cassandra would fail under heavy load, when Java isn't allowed to open the required number of file descriptors.

Hadoop Hadoop is another good example where a large number of Hadoop, OS and application parameters need to be tuned for best performance on a specific cluster. Number of MapReduce slots, as well as the Java heap size were the first configuration parameters to be tuned. The goal was to maximize resource (CPU, Memory) utilization of the cluster and Hadoop default values are usually too small. Configuring Java heap size is dependent on available memory as well as application characteristics. It was observed that some applications failed to run without sufficiently large heap size/memory and needed to be tuned properly. Several other OS and Hadoop configuration parameters were tuned, including, but not limited to, increasing HDFS block size, increasing `TimeOut` value and sorting parameters like `io.sort.factor` and `io.sort.mb`. More details on Hadoop tuning can be found in prior work [2, 3, 8, 12].

In addition to Hadoop cluster configuration, applications require tuning for best utilization of resources. For the experiments reported in this paper, the number of reduce tasks were configured to take advantage of available reduce slots in the cluster.

Redis Tuning Redis was relatively straightforward. The default configuration file was used, and the limit on concurrent open sockets was increased to 15000 (via ulimit). It was found that a single Redis process cannot saturate the Xeon E5-2690 server. Further process scaling experiments suggested that at least 4 Redis instances were needed to saturate the server.

4.3 Macro Architectural Characterization

This section presents a number of plots for system wide characteristics of application behavior that were generated using colplot. Colplot results for all the benchmarks, for each of the four macro-architectural categories are presented. Analysis of the results show that for a particular benchmark or phase in a benchmark, only *one* of the four categories is interesting. Table 3 provides a legend for understanding the information conveyed by these figures.

4.3.1 Memcache

Memcache was studied in a one server, four client configuration. The Memcache server was started with the parameters described previously in Table 2. A number of key-value combinations were studied, but in the interest of space, results will be reported for key and value sizes of 32 bytes and 25 KB respectively. This particular value corresponds to the *Image Preview* distribution described in [33].

CPU Intensity A well-tuned version of Memcache isn't very compute intensive. Although, if the number of worker threads is made extremely high, the number of context switches between the threads becomes high, and the workload becomes *artificially* compute bound, since most of the time is spent context switching, rather than doing useful work.

DRAM Intensity Since Memcache is essentially an in-memory cache, it is *always* bounded by DRAM capacity, irrespective of the size of key value pairs. Although, for server class machines considered in this study, DRAM bandwidth was *never* the bottleneck – the maximum observed DRAM bandwidth utilization was only 9.3% of peak bandwidth.

Table 3: ColPlot legend

| CPU	
User	Percent time spent in user mode (application)
Sys	Percent time spent in system level (kernel)
Wait	Percent time spent idle waiting for an IO request
Memory	
Map	Combines mapped and anonymous memory
Buff	Memory used in system buffers
Cached	Memory in the pagecache, not including SwapCache
Slab	memory allocated to slabs
Inact	memory allocated to process that is no longer running
I/O	
ReadMB	Reads to disk in megabytes
WriteMB	Writes to disk in megabytes
Network	
InMB	Data sent, in megabytes
OutMB	Data received, in megabytes
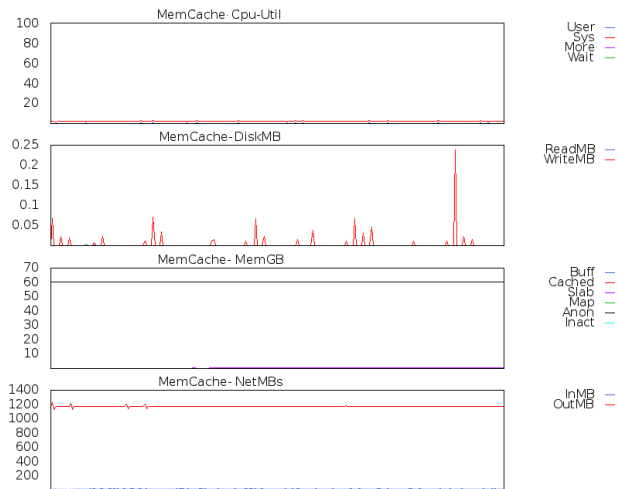
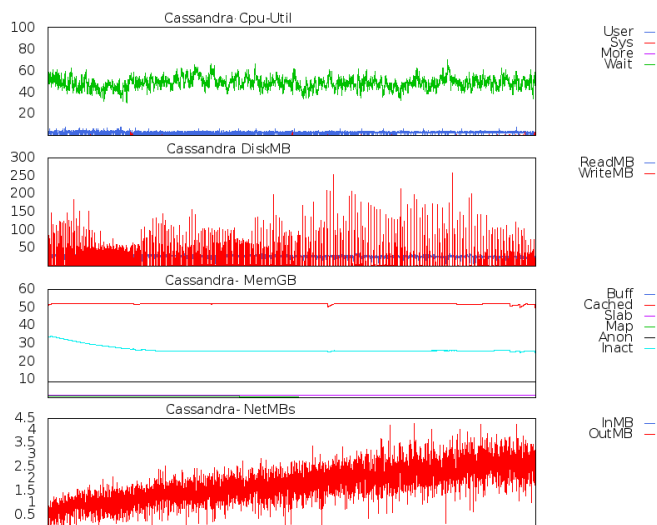**Figure 4: System-wide macro-architectural performance for Memcache.**

Disk I/O Intensity Memcache displays close to negligible I/O activity, since all the data is made to fit in the DRAM by design.

Network Intensity Depending on the use case, the line rate (1.25 GB/s) of a 10 Gb ethernet card was achieved in a number of experiments. This tends to be especially true for use cases with larger value sizes, where the relative cost of processing and sending the data over ethernet is amortized over the larger sizes of data being transferred.

4.3.2 Cassandra

The primary bottleneck of a database has traditionally been the server I/O subsystem. NoSQL databases are no different. For Cassandra characterization, the YCSB [25] client was used, which allows users to drive the database with a number of different workloads (workload A - workload F), each varying in the proportion of read, write, insert, read-modify-write, and scan operations. The general profile

Figure 5: System-wide macro-architectural performance for Cassandra.

Figure 6: Task execution time for all Map/Reduce tasks in Nutch Indexing.

of Cassandra doesn't change much with the YCSB workload under consideration. It is almost always bottle-necked by I/O, although the degree of saturation varies with the workload. The results for YCSB scan-intensive workload E are illustrated in Figure 5. Next, the characteristics of the workload as a function of previously identified macro-architectural parameters are explained.

CPU Intensity Most of the CPU activity is caused by the server threads waiting for I/O to complete. This can be clearly seen in the CPU summary graphs, where the green part corresponding to *CPU Wait time* dominates – on an average the CPU spends close to 50% of the time waiting for I/O. The rest of the CPU activity is equally divided between the user and kernel. This CPU profile remains similar for other YCSB workloads as well, but the percentage of CPU time spent waiting for I/O is reduced for read intensive workloads, since some of the requests hit in the pagecache of the server DRAM.

DRAM Intensity Cassandra is never bound by DRAM bandwidth, although the servers utilize most of its DRAM capacity, the majority of which is for caching I/O requests in DRAM by the OS. This can be seen in Figure 5, where the most of the DRAM capacity is used by *Cached* and *Anon* types of pages. Some DRAM capacity is also used up by Java, but has not been garbage collected, as indicated by the *Inact* part of the DRAM usage.

Disk I/O Intensity Disk/storage is by far the most exercised system in the Cassandra server. Workload E is a scan intensive workload that requires a lot of random reads from the disk. This behavior leads to a very consistent read throughput of about 50 MB/s from the hard disk drive. Due to the internal commit and compaction operations of Cassandra, spikes in write to disks are seen, which are evidenced by the periodic red spikes in Figure 5. Since the window of time is large (2.5 hours), the write traffic spikes appear as a contiguous phenomenon. Zooming in to the data reveals the correct behavior of the application.

Network Intensity Cassandra under the current test configuration doesn't exercise the network by much. For
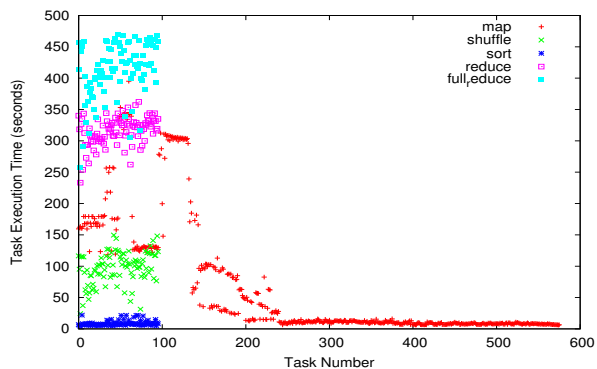
workload E, the network traffic is especially small since scan operations cause a large amount of disk I/O, while the amount of data that has to be sent back to the client is small. Hence, the maximum network utilization is around 1% of the peak available bandwidth. For test cases involving multiple clients and read intensive workloads, this utilization becomes better, but doesn't come close to saturating the line rate.
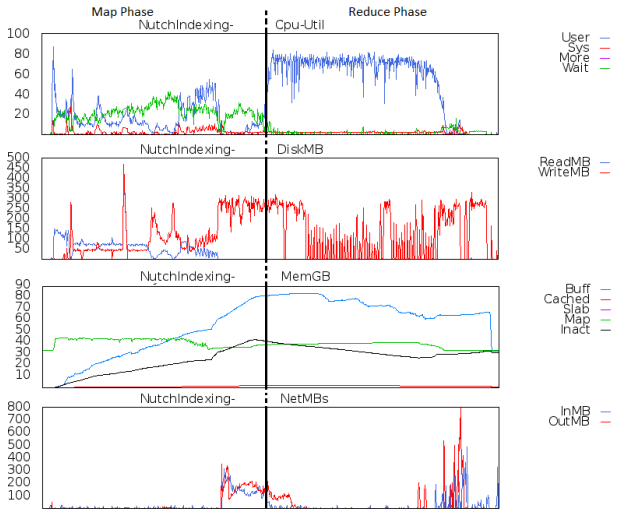
4.3.3 Nutch Indexing

The indexing sub-system of Nutch was studied as included in the HiBench[31] suite. The workload uses the default Linux dictionary file and automatically generates web crawl data where hyperlinks and words follow the Zipfian distribution. Crawl data is read and decompressed during map phase and is converted into inverted index files in the reduce phase. The dataset considered is 10M pages large. Several Map/Reduce parameters were changed to improve performance:

- MapReduce slots; tuned according to CPU count
- Status report timeout; increased to avoid killing jobs pre-mature
- Number of Reduce jobs; originally set to 1, increased to utilize all CPUs
- Map/Reduce Heap size; tuned for DRAM capacity
- Overlap between Map/Reduce jobs; reduced to eliminate undesired impact

Figure 6 shows execution times for all Map/Reduce tasks including execution time of shuffle and sort. Map execution times vary greatly as is expected considering the varying input file sizes and formats. Shuffle time is short, since we reduce Map /Reduce overlap and shuffle starts only after most map jobs have finished. Sort time is insignificant, and Reduce jobs take noticeable time which justifies increasing number of reduce tasks to the highest number of concurrent tasks available on the cluster.

Figure 7 shows system behavior (CPU, disk, memory and network) of a single datanode when running Nutch indexing on the Hadoop cluster.

CPU Intensity: During the map phase, where crawl data is read and decompressed, most CPU cycles are spent waiting for I/O. On the other hand, the reduce phase, where inverted indexes are generated , is very compute intensive.

Figure 7: System-wide macro-arch performance for Nutch indexing.

Figure 8: Execution time sensitivity to total DRAM capacity per node (64 GB vs. 128 GB) for Nutch indexing.

with up to 80% CPU utilization. Scaling experiments suggest that enabling simultaneous multhithreading and utilizing all logical threads (32) in both map and reduce provides the fastest total execution time. Increasing Map/Reduce slots beyond that had no benefits.

DRAM Intensity: An extensive set of memory capacity sensitivity studies were done and it was identified that 1.2 GB memory per map is the optimal capacity for map execution for Nutch. Capacity below that caused failures due to out of heap errors. Experiments also showed undesirable impact of concurrent map/reduce jobs due to resource sharing, including DRAM. To address that, overlap between phases was decreased by increasing the "slow-start" parameter in Hadoop configuration. Figure 8 shows MapReduce execution time sensitivity to DRAM capacity (64 GB vs. 128 GB). As can be seen, increasing the memory capacity to 128 GB from 64 GB leads to reduction in the average execution time per map by about 5×.

Disk I/O Intensity: Nutch reads and transforms a large amount of data leading to high disk read throughput during map, and high write throughput (up-to 250 MB/s) during shuffle and reduce phases. The compression option used on input and output data can impact CPU and Disk I/O intensity.

Network Intensity: Activity on network happens at two distinct phases of execution: at the end of map where all data is shuffled across nodes to prepare for the reduce phase, and also at the end of reduce phase where the inverted index files go over the network to be written back to HDFS. The map phase was mostly data-local which explains the low network activity during that phase. Higher network activity is expected on larger cluster with hundreds or thousands of nodes.

4.3.4 Data Analytics

The Data analytics workload in CloudSuite [27] uses the existing implementation of bayes classification algorithm in Mahout for categorizing documents. Bayes classifier is a simple probabilistic classifier based on the bayes theorem. The
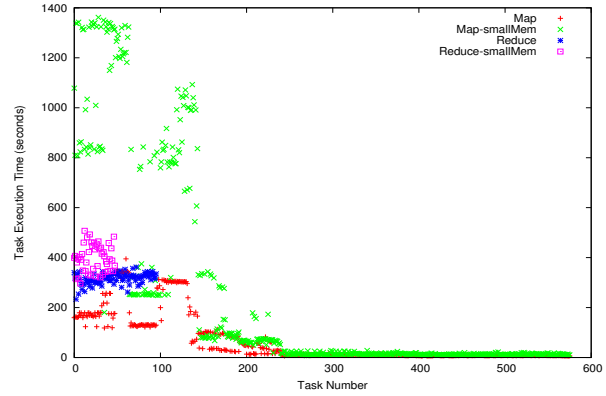
input dataset for categorization is the entire English language Wikipedia articles database (45 GB), which is freely available for download from the Wikipedia Foundation. The training dataset that is used to train the classifier is a subset of the Wikipedia articles (9 GB). This workload represents an offline mode of data analytics, where the training data is used to build the classifier model first, and the input dataset is subsequently used for categorization based on the classifier model. There is no feedback loop to the classifier model. The workload is divided into four phases, in addition to saving the initial Wikipedia-dataset to HDFS.

- The first phase splits the input and training Wikipedia dataset with a pre-defined chunk size across the available nodes in the Hadoop cluster.
- The second phase organizes and further splits the input data (from phase 1) based on predefined categories.
- The third phase builds a classifier model based on the training dataset using the Bayes algorithm. The classifier model is built once within the workload.
- The fourth or classification phase runs each input data split created in the earlier phases against the classifier model to categorize data. Most of the analysis of results in this section pertains to the classification phase.

The results of this study focus entirely on optimizing the execution time of the workload. Figure 9 shows the system-level characterization across the full lifetime of the workload.

CPU Intensity: The classification phase, which categorizes the input dataset, is highly CPU intensive, driving CPU utilization close to peak, as shown in phase 4 of Figure 9. An extensive set of sensitivity and scaling experiments were conducted by mapping map/reduce tasks to varied CPU configurations in each node. Figure 11 shows a subset of map execution time for different concurrent map configurations (8, 16, and 32) per node across the classification phase of the workload. It can be observed that increasing concurrent map tasks past sixteen, leads to 2× increase in execution time per map task. This is due to physical CPU limit of sixteen on the dual-socket Xeon node, even though it supports thirty two threads using hyper-threading. It can be concluded that if shortest execution time is desired, hyper-threading/simultaneous multi-threading (SMT) must be disabled and independent physical CPUs should be used.
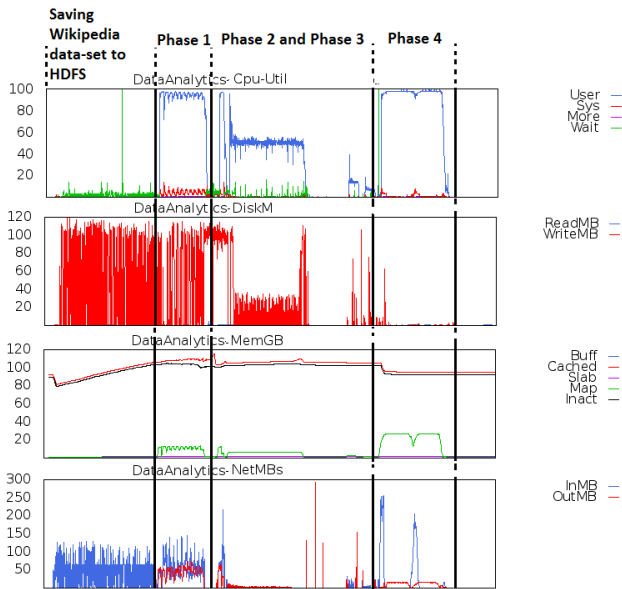
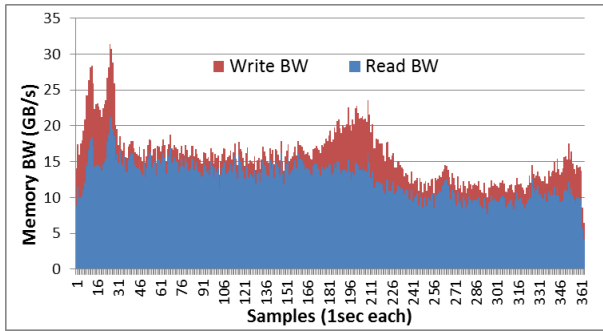**Figure 9: System-wide macro-arch performance for data analytics.**

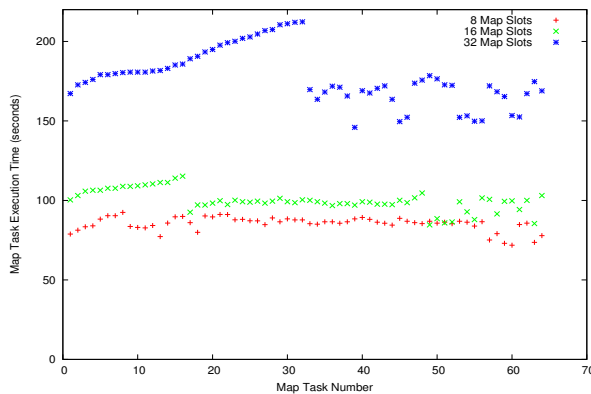**Figure 10: Aggregate memory bandwidth of classification phase in data-analytics.**

Figure 11: Execution-time sensitivity to Map slots for classification phase in data-analytics.
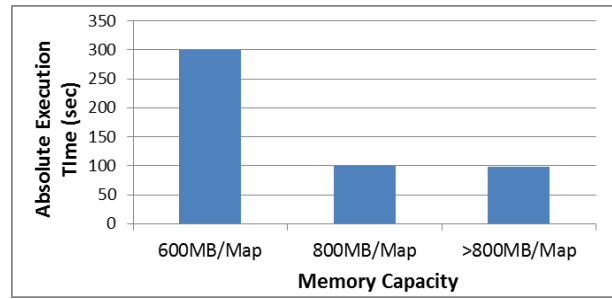

Figure 12: Data-analytics memory capacity scaling.

Although SMT is an effective architectural solution to increase throughput by executing multiple threads on a physical CPU [40], sharing of resources (execution units, caches etc.) leads to slowdown among homogeneous map jobs in a node. In order to achieve maximum map job-level parallelism and lowest time to completion, the Hadoop cluster must be provisioned to support enough independent physical CPUs for concurrent execution of all map jobs (function of the input data-set).

Memory Bandwidth Intensity: Figure 10 shows the aggregate memory bandwidth (including reads and writes) for the classification phase. Overall memory bandwidth utilization for the node is less than 20% of peak memory bandwidth. The DRAM page hit rates were also studied and they average around 80%, suggesting good locality of reference at the memory controller. Read bandwidth is also much higher than write bandwidth. Sensitivity experiments suggest memory bandwidth scales almost linearly with increasing map jobs per node up until all CPUs are fully utilized, although total bandwidth utilization remains less than 20% under full system-load.

Memory Capacity Intensity: An extensive set of memory capacity sensitivity studies were done to identify optimal capacity requirements per map job. Figure 12 shows the average map execution time with a subset of identified scaling memory capacity data points. Experimental results suggest that a minimum of 600 MB of heap space (constrained by physical memory) is required per map job for successful execution, and anything less results in heap memory failures. Increasing memory capacity per map job, 800 MB was identified as the most optimal point. The average execution time of map jobs reduces by 3× - from over 300 s to 100 s by increasing capacity from 600 MB to 800 MB. Scaling memory capacity beyond 800 MB per map does not achieve any performance benefits, as shown in Figure 12. This translates to a total of 25 GB heap space on a node with 32 CPU threads (with SMT enabled). In comparison, each dual socket Xeon server node can support up-to 768 GB of DRAM.

Disk I/O and Network Intensity: Most of the disk I/O activity is recorded in saving the initial Wikipedia dataset on HDFS and the first phase of the workload execution, which includes splitting and writing the updated input and training data on HDFS, as shown in Figure 9. Network traffic is also most prevalent in this phase. The classification phase has intermittent disk and network I/O.

4.3.5 Redis

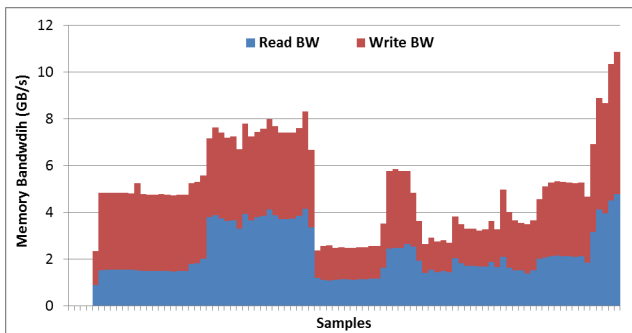Redis supports a rich set of data structures and provides various commands to manipulate them. While Redis can

35

Figure 13: Redis memory bandwidth.

Figure 14: System-wide macro-arch performance for Redis.

also be used as an in-memory cache layer (like Memcache), this paper reports benchmarking results for a set of more complex instructions that are more relevant to the data analytics use-case. Specifically, this paper's benchmark tests utilize lists, sets, and sorted sets, which have all been confirmed to be used extensively in several real-world Redis deployments [19]. Lists are often used to implement job queues and maintain timelines. Sorted sets are naturally used in dashboards, leader boards, and priority queues. Sets are commonly used to maintain unique item lists which is a common task in several real-time analytic applications [5].

Figure 14 shows the macro-architecture characterization of our Redis benchmark. The workload comprises of 5 stages, that are easily distinguishable in the figure: (1) *lpush* inserts new elements to the head of a list. (2) *lpop* returns elements from the head of a list. (3) *zadd* adds an element with a specified score to a sorted set structure. (4) *zrange* - returns a specified range of elements from a sorted set. The workload is set to return the element with the minimal score. Lastly, (5) *sadd* adds an element to Redis set structure. (A set holds an unordered collection of unique elements.)

Network Intensity As can be seen in Figure 14, the server is clearly limited by the network bandwidth: it saturates the 10GbE network link throughout the entire run, while all other resources are lightly used. (lpush, zadd, and sadd saturate the ingress direction, while lpop, and zrange saturate the egress direction).

CPU Intensity CPU load is low for Redis. Even for more compute-heavy operations like zadd and sadd, the network bandwidth is getting saturated without fully utilizing the cores. Moreover, note that the core utilization in Figure 14 is reported for the 4 active cores only. Redis is a single-threaded workload, and 4 Redis instances proved to be enough to saturate the network in all experiments. This means that the majority of cores in the system are practically left idle.

Disk I/O Intensity Since Redis stores all data in memory and serves all requests from memory, there in no meaningful disk traffic. While Redis periodically writes data back to hard-drive to preserve data persistence, this turned out to have minimal performance effect in our experiments.

DRAM Intensity As can be seen in Figure 13, DRAM bandwidth utilization is low (less than 10 GB/s corresponds to less than 10% utilization) throughout the entire run. However, since Redis stores all data in memory, the server memory capacity dictates the maximal database size the application can store. To illustrate the DRAM capacity pressure,
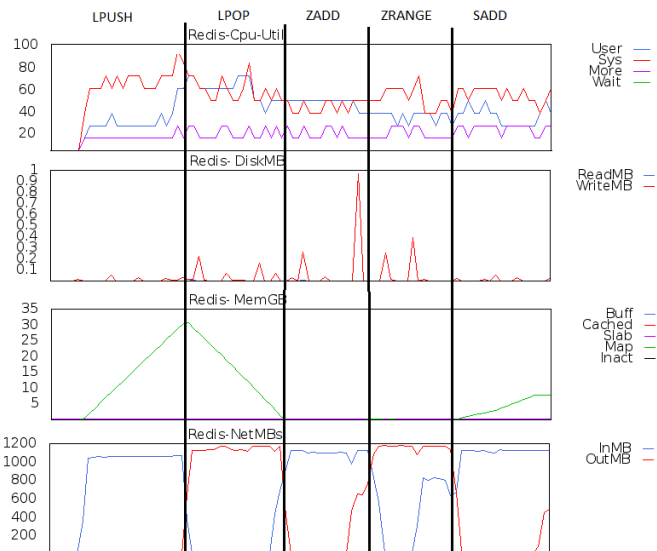
the *lpush* phase stores 32 GB of data to a Redis list structure. The increase in memory capacity can be clearly identified in the memory usage plot in Figure 14. The *lpop* phase later removes all elements from the list, and it can be seen that the memory usage is indeed going down. (The score range is kept limited in the *zadd* phase so the set size is kept minimal. Further increase in the range in the *sadd* test results in an increase of memory capacity usage.)

5. RELATED WORK

There exists a lot of relevant literature on a number of Big Data applications and benchmarks. This section provides an exhaustive list of the most relevant work. The related work is broken down into two main sections: (i) Microarchitectural characterization of servers, and (ii) Client side analysis. A highly specialized characterization work has resulted from the growing popularity of the Hadoop MapReduce framework, focusing on analyzing the system performance from a "job" perspective. Section 5.3 has been dedicated to discussing the relevant job-level characterization efforts.

5.1 Client Side Analysis

Most of the characterization efforts from the industry have focused on comparing client-side performance of products that offer similar features. Netflix uses Cassandra deployed in AWS [10, 13] as a key infrastructure component of its globally distributed streaming product. They have conducted extensive characterization of Cassandra [7, 20]. Most of these studies focused on scalability aspects of Cassandra and measuring client-side throughput in terms of requests serviced per second, as well as client-side response latency. DataStax, one of Cassandra's commercial vendors has provided an extensive study on client side comparisons for many NoSQL databases [9]. This work was an independent extension of the work done in [36] which provided client side comparisons for a number of leading NoSQL databases for one particular use case – application performance management

(APM). Similar scaling studies have been done for Memcache at Facebook [38, 35].

5.2 Server Microarchitectural Characterization

CloudSuite [27] characterization was one of the first works that was done in the field of designing benchmarks for scale-out datacenters. The authors did a great job of identifying representative applications and putting them together as a part of the benchmark suite. They also provide a comprehensive analysis of the workloads, providing results for a number of microarchitectural parameters like L1 and L2 cache hits/misses, instructions per cycle (IPC) and memory/DRAM utilization. However, their analysis stops at the boundary of DRAM, providing very little insight into the I/O behavior of workloads. Given that most applications in the domain are I/O bound, we believe that concentrating workload characterization to just the CPU-Memory subsystem doesn't provide a comprehensive picture of benchmark behavior.

BigDataBench [29, 41] provides a collection of another set of applications that represent datacenter workloads. This work is built on [27] by providing a more comprehensive list of workloads, that cover a broader spectrum of datacenter based services. They also provide representative datasets as a part of the benchmarking suite. The workload analysis done in both these papers has been extremely thorough, and has dealt mostly with analyzing the workloads from the CPU-centric perspective. The analysis has focused on metrics like IPC, MPKI and LLC hit and miss ratios, while not worrying about the client side results.

5.3 Job Level Analysis

A number of other papers have done more comprehensive, per-node analysis of targeted workloads especially for the Hadoop MapReduce framework. However, almost all prior work has focused on carrying out rigorous analysis of *one* subsystem for every workload that they have considered. Abad and Roberts [22] provide a detailed analysis of the storage subsystem for Hadoop based MapReduce jobs. For the Hadoop framework, a number of studies have carried out detailed analysis of the utilization of individual nodes, and of the Hadoop cluster in general. Ren et al. [37] provide a task and job level analysis of a production Hadoop cluster from Taobao Inc.

6. CONCLUSIONS AND FUTURE WORK

This paper presents the authors' experiences and learnings in selecting, setting up and tuning workloads from various published benchmark suites, as well as a few others based on popularity and use cases. The biggest learning that was made from these studies was the fact that a "Big Data benchmark" is a misnomer. The performance of a datacenter application is highly dependent on the test setup, which comprises of a number of moving parts, including, but not limited to the hardware configurations of the server and the client, the characteristics of the network and the fine tuning of the operating system and application settings. Without properly tuning most of the said components, the relevant subsystems of the server will not be stressed.

Secondly, in order to emulate real-world use and deployment cases, it is very important that the application is optimized for providing best performance on the available resources. This process results in most representative application behavior and characteristics, leading to effective provisioning decisions for each server node.

Even when the application characteristics for the use cases are known, deciding resources for each of the different tiers of a Web 2.0 datacenter is a hard problem. The intent of the designers is to get the maximum performance out of the available resources, be able to handle peak load without buckling under the stress, as well as making sure that the servers are not extremely over-provisioned. Knowing the bottlenecks of different applications will help datacenter architects make the right provisioning trade-offs for each tier of the datacenter. Having a "pressure point matrix" similar to Table 4 will greatly help in making such decisions. For example, if the datacenter is projected to have a big in-memory caching or analytics tier, it would be best to provision servers with adequate network resources, maybe even at the cost of direct attached storage for each server node in that tier.

Furthermore, being able to bin applications according to their pressure points, in a fashion similar to Table 4 would help decide their co-location potential across different tiers of the datacenter. For example, some applications have strict QoS requirements (the caching tier), while others may have relaxed requirements. Knowing the pressure points of different applications, will help make intelligent co-location decisions for different applications. This will lead to designing policies that will result in better utilization of resources across different tiers. We leave such studies for future work.

Table 4: System level bottlenecks for each of the applications under consideration

Workload	Pressure Points
Memcache	DRAM, Network
Cassandra	Disk
Redis	Network
Nutch (Hadoop)	CPU, Disk
Data Analytics (Hadoop)	CPU

References

[1] Hadoop. `http://hadoop.apache.org`.

[2] Intel Distribution for Apache Hadoop Software: Optimization and Tuning Guide. Technical report, Intel.

[3] Optimizing Hadoop Deployments. `http://www.intel.com/content/dam/doc/white-paper/cloud-computing-optimizing-hadoop-deployments-paper.pdf`.

[4] Redis. `http://redis.io/`.

[5] Sensor Andrew. `http://sensor.andrew.cmu.edu`.

[6] Cassandra at Twitter Today. `https://blog.twitter.com/2010/cassandra-twitter-today`, 2010.

[7] Benchmarking Cassandra Scalability on AWS - Over a million writes per second. `http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html`, 2011.

[8] Hadoop Performance Tuning Guide. Technical report, AMD, 2012.

[9] Benchmarking Top NoSQL Databases. Technical report, DataStax Corporation, February 2013.

[10] Big Data Analytics at Netflix. Interview with Christos Kalantzis and Jason Brown. `http://www.odbms.org/blog/2013/02/big-data-analytics-at-netflix-interview-with-christos-kalantzis-and-jason-brown/`, 2013.

[11] Data Caching. `http://parsa.epfl.ch/cloudsuite/docs/data-caching.pdf`, 2013.

[12] Hadoop Job Optimization. Technical report, Microsoft IT SES Enterprise Data Architect Team, 2013.

[13] Nosql at netflix. `http://techblog.netflix.com/2011/01/nosql-at-netflix.html`, 2013.

[14] Apache Nutch. `http://nutch.apache.org`, 2014.

[15] Apache Spark – Lightning-fast cluster computing. `http://www.odbms.org/blog/2013/02/big-data-analytics-at-netflix-interview-with-christos-kalantzis-and-jason-brown/`, 2014.

[16] Collectl. `http://collectl.sourceforge.net`, 2014.

[17] Collectl Utilities. `http://collectl-utils.sourceforge.net`, 2014.

[18] In-Memory Computing Market worth \$13.23 Billion by 2018. `http://www.researchandmarkets.com/research/btkq7v/inmemory`, 2014.

[19] Redis Roundup: What Companies Use Redis? `http://blog.togo.io/redisphere/redis-roundup-what-companies-use-redis/`, 2014.

[20] Revisiting 1 Million Writes per second. `http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html`, 2014.

[21] Who's using Redis? `http://redis.io/topics/whos-using-redis`, 2014.

[22] C. Abad, N. Roberts, Y. Lu, and R. Campbell. A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns. In *Proceedings of IISWC*, 2012.

[23] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of SIGMETRICS*, 2012.

[24] companiesandmarkets.com. Global web 2.0 data center market. `http://www.companiesandmarkets.com/`, 2014.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SoCC*, 2010.

[26] F. X. Diebold. Big Data Dynamic Factor Models for Macroeconomic Measurementand Forecasting. In *Eighth World Congress of the Econometric Society*, 2000.

[27] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of ASPLOS*, 2012.

[28] B. Fitzpatric. Memcached - A distributed memory object caching system. `http://memcached.org/`, 2014.

[29] W. Gao, Y. Zhu, Z. Jia, C. Luo, L. Wang, J. Zhan, Y. He, S. Gong, X. Li, S. Zhang, and B. Qiu. BigDataBench: a Big Data Benchmark Suite from Web Search Engines. In *Proceedings of ASBD*, 2013.

[30] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, Dec. 2008.

[31] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshops*, 2010.

[32] R. Khare and D. Cutting. Nutch: A Flexible and Scalable Open-source Web Search Engine. Technical report, CommerceNet Labs, 2004.

[33] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of ISCA*, 2013.

[34] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out processors. In *Proceedings of ISCA*, 2012.

[35] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of NSDI*. USENIX, 2013.

[36] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735, 2012.

[37] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou. Workload characterization on a production hadoop cluster: A case study on taobao. In *Proceedings of IISWC*, 2012.

[38] P. Saab. Scaling memcached at facebook. `https://www.facebook.com/note.php?note_id=39391378919`, 2008.

[39] J. Taylor. Facebook scale and infrastructure. `http://http://new.livestream.com/ocp/winter2013/videos/9511147`, 2013.

[40] D. M. Tullsen, S. J. Eggers, and L. H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of ISCA*, 1995.

[41] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. BigDataBench: A Big Data Benchmark Suite from Internet Services. In *Proceedings of HPCA*, 2014.

[42] A. Wiggins and J. Langston. Enhancing Scalability of Memcached. Technical report, Intel, May 2012.