Efficient In-memory Data Management: An Analysis

Hao Zhang†, Bogdan Marius Tudor†, Gang Chen§, Beng Chin Ooi†
†National University of Singapore, §Zhejiang University
†(zangh,bogdan,ooibc)@comp.nus.edu.sg, §cg@cs.zju.edu.cn

ABSTRACT

This paper analyzes the performance of three state-of-the-art systems for in-memory data management: Memcached, Redis and the Resilient Distributed Datasets (RDD) implemented by Spark. By performing a thorough performance analysis of both analytics operations and fine-grained object operations such as set/get, we show that neither system handles efficiently both types of workloads. For Memcached and Redis the CPU and I/O performance of the TCP stack are the bottlenecks – even when serving in-memory objects within a single server node. RDD does not support efficient look-up of random objects, relying on sequential scan and thus hitting the memory bottleneck. Our analysis reveals a set of features that a system must support in order to achieve efficient in-memory data management.

1. INTRODUCTION

Given the explosion of Big Data analytics, it is important to understand the performance costs and limitations of existing approaches for in-memory data management. Broadly, in-memory data management covers two main types of roles: (i) supporting analytics operations such as iterative algorithms and (ii) supporting storage and retrieval operations on arbitrary key-value objects.

This paper focuses on three such popular systems: Memcached [3], Redis [4] and RDD [6], and proposes a thorough performance analysis of both analytics and key-value object operations. The insights revealed by our analysis are as follows:

1. Analytics operations achieve poor performance due to the architecture of Memcached and Redis, that forces any data exchange to use TCP network sockets. In contrast, Spark can perform analytics operations directly on the in-memory RDD and achieve 17-50× better performance than Memcached or Redis when executing the PageRank algorithm (Section §3).

2. We discover surprising inefficiencies in the way Memcached and Redis operate with small key-value objects. The main culprit for such behavior is the TCP performance. Both Memcached and Redis utilize the efficient epoll Linux mechanism that can parallelize the servicing of key-value object requests, and do achieve high CPU utilization. However, by analyzing the CPU activity down to pipeline and cache utilization, we observe that the cores’ pipeline is empty for more than 70% of the execution time (i.e. the cores incur stall cycles) for Memcached and Redis. Unexpectedly, this is not due to the overhead of fetching the key-value objects from memory, but rather due to the code branch mispredictions and instruction cache misses. We measure an instruction cache miss rate of 20% (Memcached) and 15% (Redis) and around 50% branch misprediction rate (both). To put these numbers in perspective, a memory-bounded high-performance computing program achieves 0.01-0.05% instruction cache miss rate and less than 0.01% branch misprediction rate on the same hardware. Such important differences are attributed to frequent jumps among the control flow of the server code, the epoll mechanism and the TCP stack inside the Linux kernel. (Section §4).

3. RDD does not natively support fine-grained operations on arbitrary key-value objects. We find it impossible to modify RDD to support set operation. The throughput of get operation is limited by the lack of indexing or hashing within an RDD partition, which requires a sequential scan through all the records. Thus, RDD quickly hits the memory bottleneck. Additionally, it is hard to scale the object get throughput of RDD across multiple server nodes, because Spark isolates the RDDs of different jobs, thus preventing multiple drivers from accessing the same dataset. (Section §4).

Based on these performance insights, this paper proposes a set of design criteria that are essential to enhance the performance of efficient in-memory data management. (Section §5).

2. EXPERIMENTAL METHODOLOGY

Workloads setup. To test the analytics performance, we use the PageRank algorithm implemented in a Map-Reduce style. In the Map phase, we compute the contributed rank for the neighbors of every web page, and distribute this information to other nodes. In the Reduce phase, each node computes the new ranks for the local web pages based on the contributed ranks. We run PageRank for 10 iterations on the Stanford’s Google Web Graph dataset of 875,713 web pages and 5,105,039 links.

Spark naturally supports Map/Reduce computations, so we implement PageRank using the Spark APIs. We persist RDD into memory before we use it. We use Spark 0.8.0/Scala 2.9.3 with Java 1.7,0.

Memcached is a key-value store that only supports operations such as set and get. Thus, it is not straightforward to implement PageRank algorithm on top of Memcached. Our approach is to equip Memcached with API that can return all the keys and implement a driver program to do the computation. The driver uses
ManyBrain Java Memcached client [2]. Specifically, a driver program is hosted inside each Memcached server node and is used to manage its local Memcached server and communicate with remote Memcached servers. We coordinate all driver programs with a master program that directs the drivers into the map and reduce steps. The TCP protocol is used for communication between all Memcached servers and the drivers. We use Memcached 1.4.15 compiled using gcc 4.6.3 with the default settings. Figure 1 shows the architecture of the analytics operations on top of Memcached.

![Architecture of analytics programs over Memcached and Redis servers](image)

Figure 1: Architecture of analytics programs over Memcached and Redis servers

Like Memcached, Redis is a key-value store with basic set/get operations and a set of advanced functions such as pipelined operations, server-side scripting, and keys. A similar setup as Memcached is used for Redis, as described in Figure 1, termed Redis server-side. The driver uses Redis’ Lua scripts, without relying on a driver program. Thus, the PageRank processing can be done by the Redis servers via Lua scripts, without relying on a driver program. We refer to this manner as Redis server-side data analytics. We use Redis 2.6.16 compiled using gcc 4.6.3 with the default settings.

All the Memcached, Redis server and RDD worker are configured with a cache size of 3 GB; the Redis persistent storage backup is disabled during the experiments.

**Systems setup.** We use two types of systems for performance measurements:

1. **Cluster setup.** We perform scalability analysis on a cluster with 16 Intel Xeon X3430 nodes, each with 8 GB DDR3 RAM, 256 KB L1 Cache, 1 MB L2 Cache, 8 MB L3 Cache, inter-connected using 1Gbps Ethernet, and running 64-bit Linux kernel 2.6.18.

2. **Single-node setup.** For efficiency analysis, we need to remove the I/O bottleneck imposed by the network, and perform the experiments inside a single server node. The node has two Intel Xeon X5650 processors, each with 12 cores, and 24 GB of DDR3-RAM. The memcached server nodes are configured with a cache size of 3 GB; the Redis persistent storage backup is disabled during the experiments.

**Metric** | **RDD** | **Memcached** | **Redis client-side** | **Redis server-side**
---|---|---|---|---
CPU utilization [%] | 3.79 | 0.22 | 0.218 | 0.21 | 0.24 | 0.29 | 0.71
System time [%] | -0.01 | -0.01 | 0.21 | 0.21 | 0.21 | 0.21 | 0.21 | 0.21 | 0.21
Systime(s) | 6.8E+07 | 4.5E+07 | 4.5E+07 | 6.4E+07 | 6.0E+07

Table 1: OS-level performance of PageRank

The most surprising result is the gap in performance between PageRank using RDD versus PageRank on top of Memcached or Redis. RDD finishes one iteration 22-48× faster than Memcached, 17-50× faster than Redis client-side and over 400× faster than Redis server-side.

Redis server-side has poor performance because the Redis server operates in a single-threaded manner. Within a long-running scripting job, the server cannot service requests from other servers. Thus, increasing the number of servers worsens the performance, due to the increasing waiting time among servers.

Memcached, Redis client-side, and RDD generally scale well with the number of nodes. All the three systems exhibit a small increase in execution time from a single node to two nodes, but from 2 to 16 nodes we observe a decrease in execution time. With large number of nodes, PageRank over RDD executes so fast that its performance does not scale anymore. However, when executed on larger datasets, RDD exhibits almost linear decrease in execution time with the number of nodes.

We conclude that in-memory data analytics over RDD, Memcached and Redis client-side scales well across nodes, but there is a large performance gap among them. In the next subsection we investigate the reasons for this performance gap.

### 3.2 Single-Node Analysis

We investigate the reasons for the large performance difference as observed in Figure 2. We ran all four programs on the single-node setup and collected important OS-level performance metrics. Table 1 summarizes the performance data. We measure the cumulative CPU utilization (average number of cores used by the programs; maximum allowed is 12), the percentage of the CPU time spent by the programs in the kernel and the number of system calls performed per second.

**Algorithm**

RDD is the only program that benefits from the multicore architecture, with the RDD worker utilizing an average of 3.8 cores. In contrast, Memcached and Redis are idle for significant amount of time. Furthermore, we notice that Memcached and Redis spend a very large amount of time in the kernel, which is further confirmed by the large number of system calls performed.
3.2.1 Memcached and Redis

Using the strace system call profiler, we see that both Redis and Memcached spend more than 98% of the system-level CPU time in a sequence of system calls related to reading and writing from network sockets. However, this does not lead to high throughput. The culprit for this is the client libraries used by our PageRank implementation, that use only one synchronous connection to a server, leading to serialization of computation with communication.

We conclude that the poor performance of Memcached/Redis when performing analytics operations is caused by insufficient concurrency when using inter-process communication via TCP. For each data object request, the PageRank driver performs two system calls, while the Redis server performs six system calls, and the Memcached server performs three system calls. As the main bottleneck is linked to the performance of the object operations via TCP, this motivates us to perform a deep analysis of suitability of TCP for transporting key-value objects, that is shown in Section §4.

3.2.2 RDD

The main reason for the good performance of RDD is that the analytics operations are performed in the same process where the RDD are resident in memory. In contrast to Memcached and Redis, RDD does not perform any system calls to access the memory, relying on direct access to the process heap. Good parallelization by using Java threads further increases the efficiency of analytics operations over RDD.

4. PERFORMANCE OF OBJECT OPERATIONS

Our analysis of the in-memory analytics operations motivates us to investigate the performance of object operations on Memcached, Redis and RDD in detail. We perform an analysis of the throughput of set/get operations on arbitrary objects. However, RDD does’nt support set operations for key-value objects. Thus, for RDD we profile only the get operation, by writing a program that performs get operations by using the *lookup* API. The experiments are conducted in both multi- and single-node configuration. The performance of set and get operations are very similar, and thus we just discuss one of them in different scenarios.

4.1 Concurrency Tuning

To make sure that the performance of the server is maximized, we perform an experiment during which we vary the number of clients, the number of network connections between clients and servers, and the number of threads of the server for Memcached (Redis is single-threaded). We then select the configuration that maximizes the throughput. All subsequent experiments described in this section are performed with these configurations as follows.

In the multi-node experiments, we set (i) up to 16 Memcached servers using four threads each, three client nodes per server node, each client node holding 10 clients, each client using a single TCP connection per server; (ii) same settings for Redis expect for the thread configuration; (iii) the default settings for RDD.

In the single node experiment we use (i) one Memcached server using 12 threads, four clients, each using 50 concurrent connections; (ii) one Redis server, three clients each with 50 concurrent connections; (iii) the RDD worker is allowed to use 12 cores with default settings.

4.2 Multi-node Throughput

We run the three systems on up to 16 nodes. After data populating phase the clients/drivers continuously performing get operations. Figure 3 shows the throughput of a get operation for small objects (10B for both key and value). We see that the throughput of Memcached and Redis scales almost linearly with the number of nodes, and is in the range of millions of requests per second. In contrast, the get throughput of RDD is three orders of magnitude lower and does not scale beyond two nodes.

![Figure 3: Multi-node get throughput](image)

The large disparity in the performance is caused by two factors: First, because of the job isolation mechanism of Spark, there is only one driver that can serve requests to all its RDDs. This architecture puts a large performance burden on the driver, that can quickly become the bottleneck when serving requests to multiple nodes. In contrast, Memcached and Redis allow an indefinite number of clients to query the same dataset. In our case, even if the driver for RDD is multithreaded, the performance plateaus after two nodes. Second, RDD does not exploit hash or index mechanisms within one partition of the RDD. Thus, a lookup operation must sequentially scan the entire partition. The long response time for a lookup operation amplifies the first performance problem, because the driver must wait for the worker to sequentially scan all the records.

Memcached and Redis scale well with the number of nodes, but the throughput per-node is still substantially below the peak network throughput: 6-17 MB/s out of 125 MB/s.

4.3 Single-node Throughput

The analysis so far suggests that the throughput of a single-node server is a serious problem, affecting both the performance of analytics operations and set/get operations on arbitrary objects.

4.3.1 Memcached and Redis

We use the single-node setup to analyze the throughput of both set and get operations as a factor of key and value size for Memcached and Redis. As the key size and value size have similar effects on the throughput, we only show the results for the value size. The results, shown in figure 4 show a very unexpected behavior. The number of requests per second is almost constant when the value size changes between 10 bytes and 1 kB. This implies that it is just as expensive to transfer a small data object as a large data object, which limits the bytes-per-second throughput. To understand the reason, we perform an analysis of the CPU utilization during the transfers. Unlike the PageRank computations, in the throughput experiments, the CPU executing the server is the bottleneck.

![Figure 4: Single-node get throughput](image)

The main reason for the good performance of RDD is that the analytics operations are performed in the same process where the RDD are resident in memory. In contrast to Memcached and Redis, RDD does not perform any system calls to access the memory, relying on direct access to the process heap. Good parallelization by using Java threads further increases the efficiency of analytics operations over RDD.

![Figure 5: RDD throughput](image)

4.4 Table 2: OS and architectural performance for different value sizes (10 bytes key)

<table>
<thead>
<tr>
<th>Metric</th>
<th>10 Bytes</th>
<th>100 Bytes</th>
<th>10 kBytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU utilization</td>
<td>Memcached</td>
<td>Redis</td>
<td>Memcached</td>
</tr>
<tr>
<td>System time [%]</td>
<td>72.45</td>
<td>57.20</td>
<td>71.70</td>
</tr>
<tr>
<td>L1D miss rate [%]</td>
<td>9.82</td>
<td>7.49</td>
<td>10.19</td>
</tr>
<tr>
<td>L2D miss rate [%]</td>
<td>9.78</td>
<td>8.56</td>
<td>9.75</td>
</tr>
<tr>
<td>L3D miss rate [%]</td>
<td>9.40</td>
<td>10.65</td>
<td>9.54</td>
</tr>
<tr>
<td>Branch miss rate [%]</td>
<td>49.90</td>
<td>52.48</td>
<td>50.00</td>
</tr>
</tbody>
</table>

Table 2: OS and architectural performance for different value sizes (10 bytes key)

We perform an analysis of the Memcached and Redis server CPU activities during the transfers of objects of different key-value sizes.
Table 2 list some important CPU performance metrics with different sizes of values, which shows two consistent trends for both Memcached and Redis, for both set/get operations:

1. Instruction cache miss rate is high for small objects and decreases with an increase in key-value size.
2. Data cache miss rate has an opposing trend.

The instruction cache miss rate of 20% for Memcached and 15% for Redis when transferring small key-value sizes are very unexpected, since the code of both Memcached and Redis does not exceed 20,000 lines of C code. Furthermore, Memcached and Redis both spend all the service time inside a main loop where they check the sockets using the epoll_wait system calls, and subsequently read/write them. To put the figures of instruction cache miss rate into perspective, a highly memory-bounded program such as SP from NASA Parallel Benchmark that is performing analytics on a penta-diagonal matrix has 0.01-0.05% instruction cache miss rate on the same hardware.

The high instruction cache miss rate has deep implications on the efficiency of using the CPU, because instruction cache misses are much more expensive than data cache misses. Modern CPU cores have multiple concurrent pipelines and operate using an out-of-order execution, which allows multiple data cache misses to be fetched from L2/L3 caches or the main memory, but only one instruction cache miss. Thus, the latency of the memory is hidden under many concurrent data cache requests, but for instruction cache misses, the full latency penalty must be incurred. Due to this, CPU appears to be active during Memcached/Redis server execution, but instead is spending more than 70% of the CPU time waiting for instructions to be fetched from memory without doing useful work.

We use perf record to gather traces of the function calls in the system. During Memcached/Redis execution, the Linux kernel spends most of the time in the TCP stack code and in the epoll mechanism. But most of the L1 instruction cache misses come from the TCP stack code: by visual inspection, we see that kernel function calls prefixed with the string tcp and _inet trigger more than 30% of L1 instruction cache misses for both Memcached and Redis when using key-value objects of less than 100 bytes.

We conclude that TCP is inefficient at handling small objects. But analytics computations such as PageRank mostly encapsulate numbers inside the key-value objects, thus generating many small objects. The implications of this aspect on the efficiency of BigData analytics on top of Memcached/Redis are discussed in Section 5.

A recent paper analyzed the performance of Memcached in web cache deployments, and also notices the unexpectedly low performance of Memcached for small object sizes [5]. They observed that TCP is the main culprit for poor performance. Our results roughly confirm such findings, but in a different setting – geared for supporting in-memory data analytics and fine-grained object operations, not web deployments.

4.3.2 RDD

The single-node performance of Spark’s object lookup operation is shown in Figure 5, for varying number of records stored in the RDD. Due to the sequential scan, the memory access time is the bottleneck of the RDD performance. This can be seen by the strong correlation between throughput loss and last level cache miss rate.

5. CONCLUSIONS AND SYSTEM DESIGN IMPLICATIONS

This paper provided a detailed analysis on the performance of Memcached, Redis, RDD for both analytics operations and fine-grained key-value object operations. We show that neither system handles both tasks efficiently. The architecture of Memcached and Redis forces the computation component of the application to use TCP to access the in-memory storage, and is proven to be detrimental to performance. RDD does not support efficient object lookup operations, and must sequentially scan the dataset, thus hitting the memory bottleneck. Our analysis provides a set of insights on designing a system that efficiently handles both analytics and fine-grained object operations.

First, a system must support fast inter-process communication (IPC) within the same node, and efficient transfer of small key-values objects. Analytics applications such as PageRank generate many such objects because they need to encapsulate numbers. For example, shared memory pages can be used for this, which is fast and efficient, but complicates the object addressing model, as two separate processes must use the same address space.

Second, IPC across nodes must still use TCP, as UDP is unreliable and may lead to errors if requests are dropped. However, it must avoid as much as possible to transfer small key-value objects, potentially by encapsulating several key-value objects within a larger meta-object.

Third, a system must support indexing/encoding for fast random access of an object, as well as concurrency and fine-grained locking of objects. Thus, when doing computations that can modify some stored objects, ideally only those objects should be locked and other objects should be available for set/get operations.

6. REFERENCES