Resolving cache interference in distributed storage due to mixed workloads

Bharath Ravi, Hrishikesh Amur, Mukil Kesavan, Karsten Schwan
Georgia Institute of Technology

March 9, 2013

Abstract

Distributed storage systems employ main memory caches to improve performance by some orders of magnitude over disk access. These large scale distributed stores simultaneously face multiple workoads, each with drastically different characteristics. Interference between these competing workloads causes inefficient or sub-optimal usage of the main memory cache, causing degraded performance. We focus on read performance and analyse the extent of such interference in common workload patterns. We develop an adaptive cache management solution to mitigate these effects.

1 Introduction

Distributed key-value stores have become a popular way to store and serve vast amounts of data scalably. These systems are expected to store many terabytes of data and serve millions of clients simultaneously, while providing low access latencies.

In achieveing this, serving requests out of main memory rather than from disk can greatly reduce access latencies by several orders of magnitude [10]. However, main memory is not always an abundant resource. Prior work [12] reports that DRAM can account for as much as 37% of operational costs. Efficient use of memory is therefore critical to good performance. Even with enough main memory available to use as a cache, interference due to competing workloads can cause poor performance.

For example, consider a workload that accesses only popular data with a high probablity. This might expect to find almost all of its data in the cache memory of the appropriate datanode. However, other workloads also access the cache at the same time. A 'uniformly random' workload that accesses a large portion of the dataset might pollute the cache by reading data items that it will never access again, thus reducing the cache space available for popular items. Worse, a workload interested in a different set of popular items might access the same datanode. In this case, if available memory is not large enough to store both sets of popular items, both workloads suffer.

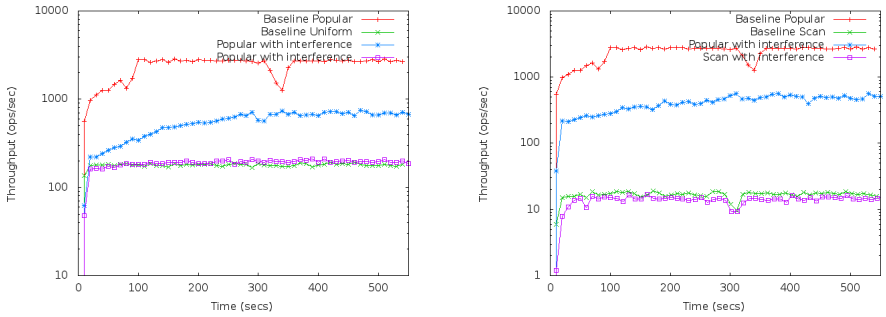
We focus on read performance of key-value stores and analyzing some scenarios of workload interference for some common classes of workloads, and develop an adaptive cache allocation solution to mitigate these effects.

We motivate our study using some practical examples of storage sytems facing mixed workloads.

1. **Scenario 1:** A big-data analysis application might run a number of MapReduce queries against a large dataset. For example a log-analysis system collects metrics and monitoring information from a large distributed system and analyzes these offline. Some of these jobs may access small portions of the dataset repeatedly: for example, jobs that are required to diagnose an ongoing performance issue. We call the stream of requests that access such popular items repeatedly a "Popular" workload. At the same time, a number of long-running jobs might be running across large portions across the dataset, to analyse performance over a long period of time. The requests made by these would be uniformly spread across all records in the dataset. We call the stream of such requests a "Scan" workloads.

   Figure 1(a) illustrates the effects of running a Popular workload together with a Scan. Compared to the baseline throughput, the Popular workload suffers by as much as 60%, while the uniform workload remains relatively unaffected.

2. **Scenario 2:** A cloud service provider using a common storage layer might see a variety of client workloads. Some clients may have a small dataset size that it is easily cacheable, while others may query large segments of their dataset.



(a) Popular with Uniform

(b) Popular with Scan

Figure 1: Interference between workloads

Further, priority clients may pay more for better performance from the back-end store. In cases deploying independent isolated stores for all clients is not scalable, a single backend store would need to ensure good performance for most of their clients.

Consider a priority client running against a small workload at the same time as other clients that access large portions of their dataset randomly. The former is similar to a "Popular" workload that may be small enough to cache within main memory and serve repeatedly. The latter are "background" workloads that together uniformly access items across a large dataset. We call this stream of requests a Uniformly Random load, or simly a "Uniform" workload.

We find that Popular workloads suffer in HBase when run together with a uniform load. Figure 1(b) highlights that performance degradation can be as much as 50% compared to the baseline throughput.

The rest of this Paper is organized as follows. Section 3 explains our system model and assumptions. Section 4 presents an analysis of workload interference in different scenarios, and Section 5 presents a brief overview of our proposed solution.

2 Related Work

A number of works [8] [9] analyze interference due to mixed workloads. Most of these, however, do so from the perspective of a single machine's hardware cache. Ghosh et. al [8] study L2 cache contention due to multiple applications and develop an optimal process-to-core mapping. Mesnier et. al [9] strike a similar chord as us, by differentating between different classes of I/O requests and using this to perform more informed cache management. However, they do so from the perspective of a single machine's Operating System cache. Further, their classes of workloads are static and unlikely to change, unlike a key-value store where a workload might change in characteristics.

On caching at the application layer, especially in large distributed systems, various systems add an additional caching layer between an underlying store and the front-end application. However, all work we found viewed client requests as a single workload, and do not consider cases where workloads could be distinguished into different classes based on theri characteristics. Memcached [7] adds an in-memory store

Fan et. al [6] discusses using a small front-end cache can provide effective load balancing across many storage nodes. While they prove that using such a cache prevents adverse loads from developing on any single node, they do not consider performance degradation to to poor management or inefficient use of available memory on nodes, or analyze interference effects that could develop in either the front-end cache or the underlying store in workloads with mixed characteristics.

TODO: The following are yet to be reviewed in detail:

1. Hotspot Prediction and Cache: Dynamically predicting and caching hotspots in a stream processor [11].

2. HashCache: A scalable network cache. This paper seems more aimed at network caches like CDNs, rather than a distributed storage system. [3]

3. Workload analysis of large scale keyvalue stores [**?**]

3 System model

We use the popular HBase key-value store [2] as a base model for our analysis. We choose HBase for anumber of reasons. Firstly, HBase is a popular commercial key-value store used in many data intensive applications **TODO: quote examples.** Secondly, the main memory record cache within HBase is characteristic of a large number of other distributed stores: it stores blocks of records and uses a modified LRU algorithm to maintain the cache. **[TODO: quote other key value stores that use simple key value: Cassandra, Redis, Voldemort**

HBase is modelled on Google's "BigTable" [4] and is a scalable, structured "Column store". It uses the Hadoop Distributed File System (HDFS) [1] as an underlying filesystem which provides a reliable, available raw filesystem.

TODO: Detailed review of other components of HBase architecture

4 Analysis of Intereference effects

4.1 Experimental Setup

Testbed: Our testbed consists of 8 physical machines, each with 2GB of main memory and a 10GB disk. Each node is connected to all others through a network with a 1Gbps interconnect between any pair. Each machine runs a HDFS datanode and a HBase regionserver. We configure HDFS to use a replication factor of 1. HBase reads records from HDFS in "blocks" of size 64KB each. Each block therefore contains approximately 20 records. Each HBase node is provided with a cache space of 395MB, which is sufficient to cache approximately 6300 blocks per node. The HDFS and HBase servers are configured to use a 100 handler threads per node, which is much higher than the maximum number of client request threads.

To run the load generator, we use a single machine with identical configuration as above. We ensure that this client node is not a bottleneck in terms of CPU, memory, network or disk resources. We use three classes

Methodology: We use the YCSB benchmark [5] to populate and query the HBase dataset. The dataset consists of 22 million records of size 3.1 KB each. This translates to around 65GB of disk space, spread across the 8 disks. Against this dataset, generate workloads that simulate the Popular, Uniform and Scan characteristics. We use three different configurations:

1. **Popular Read:** To simulate clients that are interested only in a popular portion of the data, we use a Popular workload that was restricted to 60 thousand records to ensure that a significant portion of the popular data fit in memory. The Popular read selects its dataset of 60 thousand records from the parent dataset by uniformly random selection. Within this subset, it queries certain records more often than other.

2. **Uniform random read:** To simulate requests that access the dataset uniformly, we used a YCSB Uniform workload that randomly access any of the 22 million records with equal probability.

3. **Uniform scan:** To simulate requests that access a number of records sequentially, we use the Uniform scan workload. Each scan request selects a segment of the overall dataset to access sequentially. The segments to scan are selected uniformly randomly over the dataset.

   Table 1 details the configurations of each workload. We configure the number of operations for the workloads so that they finish at approximately the same time when run together. For the Popular workload's data set, we choose a subset of 60 thousand records. This is calculated to be the approximate number of records that can fit into the aggregate memory of all 8 nodes. Since the Popular workload selects each record uniformly randomly across the 22 million dataset, with high probability each of the 60 thousand records falls in a different block. These 60 thousand blocks of 64KB each occupy a total of 3.67GB, or 470MB per node: slightly higher than the available 395MB main memory cache per node.

   Each operation of the Popular and Uniform workloads is exactly 1 record read operation. For the Scan workload, each operation performs a number of sequential reads. This number (the scan length) is chose unformly randomly to be between 1 and the size of the dataset. For this reason, the number of operations for the scan workload is much lower than the other two.

   To isolate effects due to the application managed main memory cache, we flush the operating system's page cache every 10 milli-seconds on all nodes. We clear all caches by stopping and restarting HBase between consecutive runs of workloads. During each workload, we track the disk and memory consumption and HBase cache hit-rate on each node.

Workload	Distribution	Dataset size (No. of records)	Number of operations
Popular(P)	Zipfian	60K	400K
Uniform(U)	Uniform random	22M	300K
Scan (S)	Uniform random	22M	18K

Table 1: Workload configurations. M denotes 1 million units, and K denotes 1 thousand units.

In order to track cache stastics for individual worklods, we instrument HBase to allow "workload tagging". Each YCSB workload is assigned a unique ID that is used to track the number of blocks it accesses, cache hits and misses, and its cache footprint. Our experiments show that this adds virtually no overhead to the existing HBase implementation.

Next, we analyse how these workloads interact in each case. In following analyses, we denote the Popular workloda by the letter 'P', the uniform workload by 'U' and the scan workload by 'S'. 'PU' thus refers to a pair of one Popular and one Uniform workload run together.

4.2 Baseline

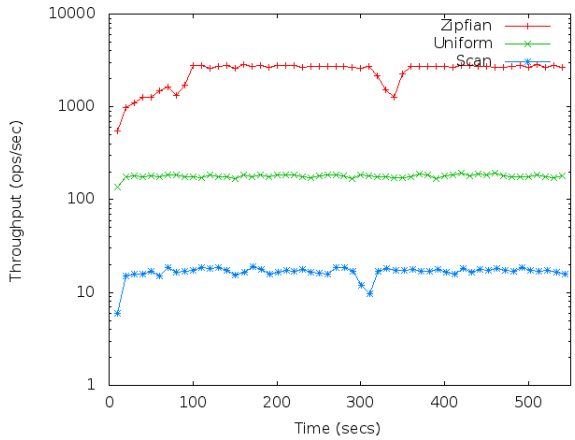


Figure 2: Baseline Throughputs

We first run each workload independently to measure our setup's baseline performance. In each run, we run the relevant workload twice: once to prepopulate the cache, and a second time to measure throughput. Each workload is run until throughput is saturated. Although our results are based on the assumption that the cache is clean/empty at the beginning, we also ran the same experiments by prepopulating the cache with the Popular workload. We found virtually no difference in results.

Figure 2 shows the base throughputs of the three workloads while Table 2 shows the number of operations per workload, the average disk utilization across all 8 nodes and the cumulative cache hit rate calculated across all 8 nodes. The cumulative cache hit rate is the total number of cache hits over the total number of cache accesses across all nodes, over the entire duration of the workload.

We observe that the throughput for Popular workload is an order of magnitude greater than the Uniform load, which in turn is greater than the Scan load.

4.3 Interference effects

Scenario 1: Popular vs Uniform. We run a Popular and a Uniform workload simulatenously to determine interference between the two. We initially warm-up the cache by running a Popular workload, to populate the cache with "popular" items followed by the two simultaneous workloads. Figure 1(a) highlights this. Table **??** contains details on the workload. We observe that disk utilization is 63.5%, indicating that the disk is not yet a bottleneck. However, we note that the cache hit rate for the Popular workload is 17.6% lower than the baseline. The degradation in throughput is around 61%.

The Uniform workload sees a 19% drop in throughput and a 2% increase in cache hit rate **TODO: Explain this increase: this is likely due to 'positive' interference with the Zipfian, that retains blocks in cache or longer**.

Scenario 2: Popular vs Scan. Similar performance degradation occurs when running a Popular workload with a Scan. Figure 1(b) and Table 4 describe this effect. We see a 68% drop in throughput of the Popular workload, along with a 15% drop in cache hit rate. The scan workload sees a 24.2% drop in throughput.

5 Preventing interference through cache throttling

The above experiments reveal that Popular throughputs are severely degraded when run together with Uniform or Scan workloads, while the latter remain relatively unharmed. Further, we note that the

Workload	Average Disk utilization (%)	Cumulative cache hit rate (%)	Avg. throughput (ops/sec)
Popular (P)	55.45	82.5	1387.8
Uniform (U)	62.50	8.3	286.20
Scan (S)	28.55	8.3	16.45

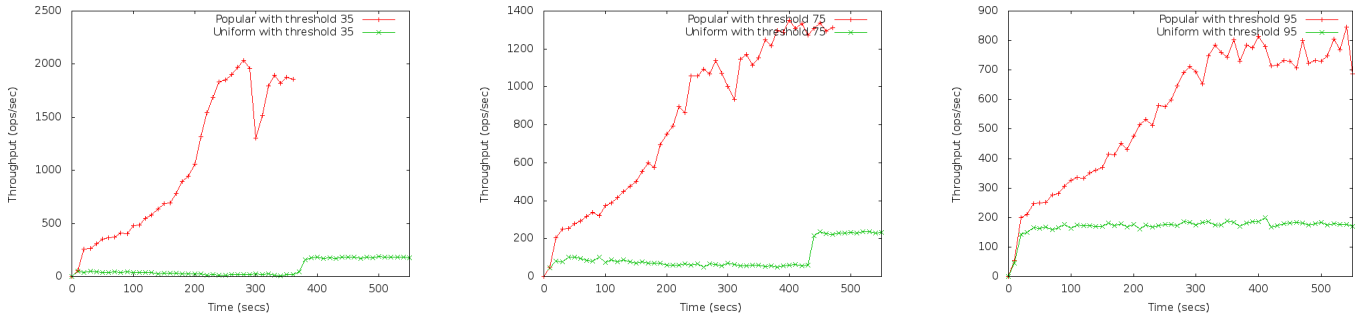
Table 2: Baseline workload statistics

Workload	Average Disk utilization (%)	Cumulative cache hit rate (%)	Avg. throughput (ops/sec)
Popular (P)	68.85	64.9	537
Uniform (U)	68.85	10.1	231

Table 3: Popular and Uniform

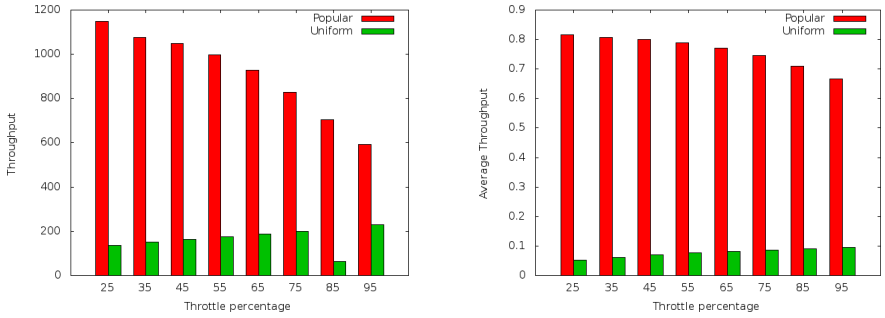
Workload	Average Disk utilization (%)	Cumulative cache hit rate (%)	Avg. throughput (ops/sec)
Popular (P)	63.59	67.47	443.64
Scan (S)	63.59	–	12.46

Table 4: Popular and Scan



(a) Threshold = 35 (b) Threshold = 75 (c) Threshold = 95

Figure 3: Effects of throttling Uniform in Popular vs. Uniform



(a) Throuhgputs vs Throttle values (b) Cache hit-rates vs Throttle values

Figure 4: Effects of throttling Uniform in Popular vs. Uniform

cache hit rates are much Explain reasoning behind ID throttling. To conclusively prove that the source of performance bottleneck is the cache, we first completely prevent one of the workloads from accessing the cache. We assign each workload a unique ID to distinguish between requests from each. We first consider the Popular vs. Uniform case, and prevent the Uniform workload from accessing the cache completely.

We perform cache throttling by preventing certain client requests from accessing the cache, thus reserving cache space and bandwidth for performant workloads. By tagging workloads with a uniqeue identifier, we distinguish between workloads that have a

higher probability of a cache hit, and reserve cache for such requests by throttling workloads with lower cache hit rates. Throttling is performed by preventing a certain percentage of the throttled workloads requests from accessing the cache. We call this percentage the 'throttle value'.

Intuitively, setting too low a throttle value severly limits the throttled workload's throughput, while setting it too high would cause interference with unthrottled workloads due to competing cache accesses. This is clear from Figure 3. With a low threshold of 35, only 35% of the Uniform loads requests are allowed to access the cache. With this,we see a high throughput of close to 1800 ops/sec for the Popular workload. However, as long as the Popular workload runs, the Uniform workload suffers severely, with an average throughput of around 20 ops/sec. On the other hand, with a high throughput of 95, while the Uniform workload is remains unaffected, the maximum throughput of the Popular load is limited to around 800 ops/sec.

Figures 4 show the effects of various values on the throughput and cache hit-rate. We see that lowering throttle value (stricter throttling) increases both throughput and cache hit-rate for the Popular load, while decreasing it for the Uniform load. Throttling thus give us a tool to trade-off between a "cache-friendly" load and a "cache-unfriendly" load, allowing us to allocate more of the cache to the workload that will make better use of the cache.

6 Next steps:

1. Now that we have throttling, a knob, develop an adaptive technique to set the right throttle value. This will be done by "sampling" a workload for a limited time and looking at its cache hit-rates and footprint size. Work loads with low cache hit rates but high footprints will throttled down more aggressively.

2. Cache throttling has no effect on interference with Scan. Look into the bottleneck for this scenario.

3. We see no interference between two Popular, two Uniform or two Scan workloads. However, two Popular workloads, each accessing a different set of items will interfere. Present results on this. The solution here is to isolate the two popular sets by moving them to different sets of region servers.

References

[1] The hadoop distributed filesystem, February 2013.

[2] Hbase: a distributed, scalable, big data store, February 2013.

[3] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. Hashcache: cache storage for the next billion. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 123–136, Berkeley, CA, USA, 2009. USENIX Association.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[6] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[7] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[8] Mrinmoy Ghosh, Ripal Nathuji, Min Lee, Karsten Schwan, and Hsien-Hsin S. Lee. Symbiotic scheduling for shared caches in multi-core systems using memory footprint signature. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 11–20, Washington, DC, USA, 2011. IEEE Computer Society.

[9] Michael P. Mesnier and Jason B. Akers. Differentiated storage services. *SIGOPS Oper. Syst. Rev.*, 45(1):45–53, February 2011.

[10] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, July 2011.

[11] Chentao Wu, Xubin He, Shenggang Wan, Qiang Cao, and Changsheng Xie. Hotspot prediction and cache in distributed stream-processing storage systems. In *Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International*, pages 331 –340, dec. 2009.

[12] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. Saving cash by using less cache. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.