Performance Evaluation in Multi-Tenant In-memory Data Grids

Anwesha Das
adas4@ncsu.edu

Abstract

Distributed key-value stores have become indispensable for large scale low latency applications like web services. Many cloud services have deployed in-memory data grids for their enterprise infrastructures and support multi-tenancy services. But it is still difficult to provide consistent performance to all tenants for time varying workloads for scaling out. Many popular key-value stores suffer from performance problems at scale and different tenant requirements. To this front, we present our study with Hazelcast, a popular open source data grid, and provide insights to contention and performance bottlenecks. We seek to answer questions pertaining to contention detection such as: the cause of contention, effect in performance with varying clients and workloads, the internals of the threading model, the concurrency model and if they have any impact on increasing clients.

Through experimental analysis, this work presents scenarios with possible ways to detect contention. Our study suggests that the presence of multiple queues of requests occurring with increasing clients is one factor contributing to a decrease in performance. Locking/unlocking data via wait/notify protocols adds to performance degradation with an increase in target throughput of clients.

1. Introduction

In-memory key-value stores have become an important building block for today's applications. Both academically and commercially, high-performance key-value stores have recently gained substantial attention. Although several such data stores exist, it is still hard to provide consistent performance to every client.

Google's BigTable [16], Yahoo's Pnut [18], Amazon's Dynamo [20], LinkedIn's Voldemort [39], HyperDex [21], Comet [23], Zynga Game's zBase [10], Cassandra [31], Memcached [22] used by Facebook [33], Twitter and GitHub's Redis [15] are popular key-value stores widely used in industry. MongoDB [7] and DynamoDB [1] have been hosted on Amazon Web Services. Google's Cloud Storage sits on its AppEngine, which uses various in-memory caching solutions like Python dataStore, MongoDB, Cassandra and RabbitMQ. Other than these giants, Cloudant [2] hosts CouchDB [3] and Joyent [6] offers Riak [8]. MemcacheD [22] in particular has been used by researchers [12, 13, 17, 24, 29, 38] recently to solve problems related to key-value stores. Table 1 shows some popular key-value stores and their users. Besides commercial users, some novel key-value stores have been contributed from academia as well such as Silt [30] and Comet [23]. It is undeniable that sustained performance is most important for tenants accessing such stores with time varying workloads.

In this paper, we present our study with Hazelcast[4], an open source in-memory data grid focusing on multi-tenancy. We seek to understand Hazelcast's performance with concurrent access from multiple clients and by varying several knobs we try to determine the source of contention. We have conducted experiments to substantiate the fact that performance degradation is indeed high with increasing number of clients. Our study suggests that multiple client end-points leading to concurrent data access is one cause. Moreover, locking and unlocking of data is another factor that contributes to overall performance degradation.

Although we have not proposed any generic solution to alleviate performance degradation in such data grids, we believe that our experimental evidences can help indicate a viable solution to improve performance in similar multi-tenant architectures.

The rest of this paper is organized as follows. Section 2 provides background about performance issues in storage systems, emphasizing key-value stores in particular. Section 3 describes the architecture of Hazelcast, its client protocol and threading model. Section 4 presents the observed experimental results in the context of multi-tenancy. Section 5 summarizes the overall results. Section 6 surveys previous efforts on efficient resource sharing in distributed storage systems. Section 7 concludes with relevant future work.

2. Background

Performance of storage systems and recently in-memory key-value stores have been the focus of research for a while. With in-memory data grids being used widely now, shared cluster storage catering to multiple tenants, still undergoes performance interference problems. Key-value stores typically have a combination of reads (get) and writes

Name	Users	Open Source
BigTable	Google	No
Pnut	Yahoo	No
DynamoDB	Amazon	No
Voldemort	LinkedIn	Yes
HBase	Cloudera	Yes
HyperTable	Ebay, Rediff, yelp	Yes
zBase	zynga	Yes
Cassandra	BestBuy, Rackspace, ebay	Yes
Memcached	Facebook, TWitter	Yes
Redis	GitHub	Yes
Hazelcast	Credit Suisse, Riverbed, Mozilla, Cisco	Yes

Table 1: Some available Key-Value Stores and their users

(put). The following characteristics present challenges to ensuring enhanced throughput with higher number of tenants and scale.

- Storage and eviction in such stores is oblivious to external factors such as which tenant, what data etc. Suppose tenant A's and B's keys share the same cluster instance, and every time tenant A's data gets evicted, then tenant A may experience low throughput.
- Since each get or put task has a very short task duration (in the order of nsecs), unlike a Hadoop job, the average task response time including scheduling/queuing etc. should not exceed a few microseconds. Executing such tasks considering consistent performance is hard.
- Every operation accesses data that was previously stored, hence *co-location of data and computation* is important in such clusters. Since multiple tenants may need to be serviced by the same instance, if that instance happens to host the required keys, it is difficult to ensure well balanced and distributed request handling.
- Prior work [13] has indicated an imbalance in data center environments, which means there are workload skews and time varying request patterns. A system coping with such fluctuations and still ensuring high throughput does encounter resource contention.
- Network contention: With an increase in the number of instances in a cluster and clients, the available network bandwidth becomes a bottleneck. Hence, performance suffers under high cluster load even with access to sufficient resources. This network inflated delay can be *overwhelming* particularly for low latency operations.
- Based on workload size and type, throughput tends to vary. Ensuring tenant performance is a challenge because of inherent workload unpredictability.

The above points intend to motivate the fact that performance interference in such low latency, high throughput compute intensive clusters is still hard to eliminate, although prior work have tried to look at it from various perspectives. Our experimental findings with Hazelcast elucidates the drop in throughput and provides insights to the cause for such degradation.

3. Hazelcast

Hazelcast [4] is an open source in-memory data grid for distributed computing. It is very popular and extensively deployed by several companies and startups. Fundamentally, Hazelcast's performance benefits through a decentralized design and its ease of deployment makes it a good choice for our study.

Since our interest is in operation handling during the transaction phase (gets/puts) and understanding what happens internally when multiple tenants simultaneously try to access similar data from the cluster, we shall limit our discussion only to the I/O threading model and the client protocol employed by Hazelcast.

3.1 I/O Threading Model

Figure 1 gives an overview of Hazelcast's threading model. Every Hazelcast instance forming a member of the cluster has three sets of I/O threads: 1 socket acceptor thread which accepts the content of the specific client's socket channel, 3 read threads, and 3 write threads. Thus, for every instance of the cluster, by default 7 threads are serving I/O operations and 5 threads handle event. As seen in Figure 1, additionally, there are dedicated threads to perform the actual operations: partition aware, generic or priority. Partition aware refers to operations that are aware of certain partitions such as `IMap.get()` and generic refers to non partition aware operations such as *IExecutorService.executeOnMember()*. For priority operations, in addition to normal work queues, there exist a partition aware priority work queue and a generic priority work queue. For further descriptions, refer to the Hazelcast [4] documentation. By default, `2 * core`

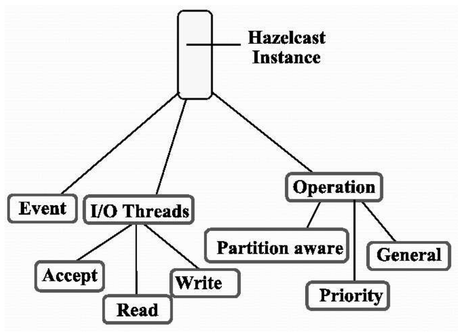

Figure 1: Threading Model: Every Hazelcast instance has multiple sets of threads to handle events and operations.

`count` is the number of operation threads started with their own queue.

Default values can be changed through configurable parameter settings. We retained the default values for our experiments and had 4 operation threads running in every instance considering 2 physical CPU cores given to each. For every client accessing a member, 11 threads can get started based on whether it is a read or a write request.

3.2 Java Client Protocol

Since we used Java clients in our experiments, we discuss the client protocol to understand how increasing clients could create contention. Every client initiates connection to the cluster through any one member via a TCP connection. The client is authenticated after which it sends periodic heartbeat messages to the cluster and retrieves a partition list. It then sends operation requests like gets/puts/inserts etc. and receives request responses. For every client, a runnable instance is created that accepts data from the socket channel and performs operations after adding tasks to the queue. Thus, with increasing client, there is a linear increase in threads that serve client requests, and the clients talk to members, the higher will be the internal sharing of data structures across multiple threads. In other words, increasing client end points increase internal resource consumption through threads and queues. Every update in cluster members is propagated to the clients and on finishing a transaction the previously opened socket is closed.

Section 4.6 shall discuss the performance impact of such multiple client end points, threads, queues, and how they affect overall throughput.

4. Experimental Evaluation

This section describes the observed performance in terms of overall throughput for varying clients. A preliminary evaluation has been conducted with Hazelcast to demonstrate contention and performance degradation with multiple clients. The YCSB [9] benchmarking tool is used to generate load on the system.

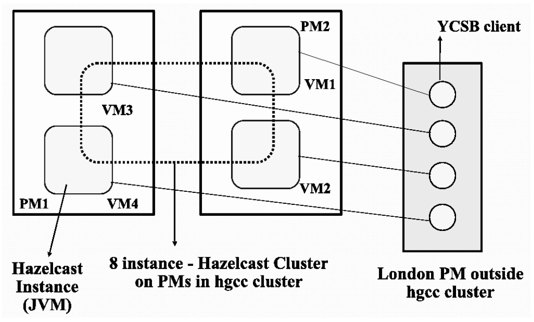
PM2
YCSB client
VM1
VM3
VM2
PM1
VM4
Hazelcast Instance (JVM)
8 instance - Hazelcast Cluster on PMs in hgcc cluster
London PM outside hgcc cluster

Figure 2: Lab Cluster Set-up: 8 instance Hazelcast cluster over 4 VMs hosted by 2 PMs.

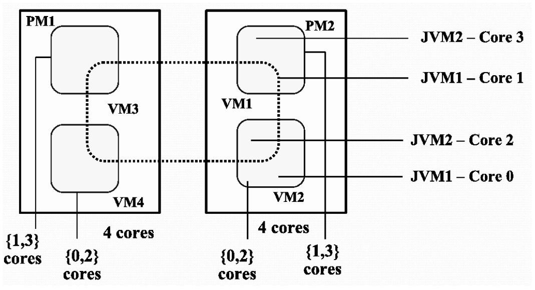

Figure 3: Set-up with 2 level pinning: Every JVM (a Hazelcast cluster instance) is pinned to a vcpu which is pinned to specific CPU cores on the host.

4.1 Evaluation Methodology

We conducted all the experiments on hgcc, a computer cluster in our lab. Each cluster node is equipped with a quad-core Xeon 2.53GHz CPU, 8GB memory, and is connected to a Gigabit network. Each host runs Ubuntu 12.04 64-bit with KVM 0.9.8. The guest VMs run Ubuntu 12.04 32-bit and are configured with two virtual CPUs and 4GB memory. Figure 2 shows our lab set-up. We set up an 8 instance Hazelcast cluster across 2 hosts and 4 VMs. Each VM ran 2 Hazelcast server instances with 2 VMs running on each host. We will use the terms *JVM* and *Hazelcast server instance* interchangeably to indicate a Hazelcast cluster instance henceforth.

A separate host the *London* (a non-hgcc machine), ran multiple instances of YCSB client to create a multi-tenant workload. The main metric used in evaluating performance is overall throughput in ops/sec.

4.2 JVM-VCPU Pinning

This section describes our observations with pinning of JVMs with a specific physical core. The idea is to prevent thread migration across cores and to reduce overall context switch overhead which might arise due to contention. However our results suggest that pinning does not have any impact on performance. Hence, our conjecture that re-spawning of threads on different cores or thread migrations may have some affect on performance turned out to be false, at-least under the limited scale of experimentation.

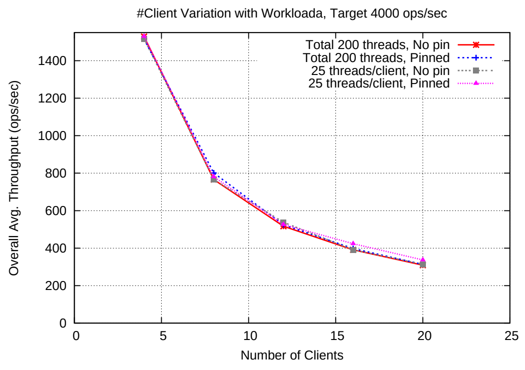

Figure 4: Comparison of JVM-VCPU Pinned versus Unpinned.

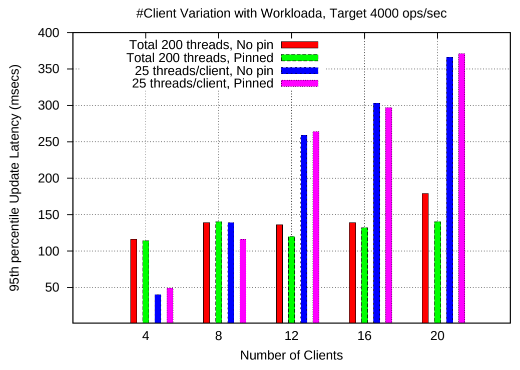

Figure 6: 95th percentile latency plot.

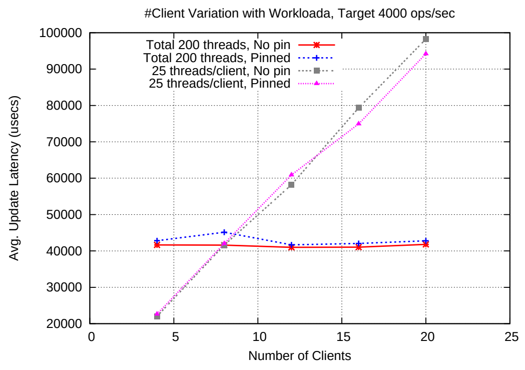

Figure 5: Increasing latency with client variation

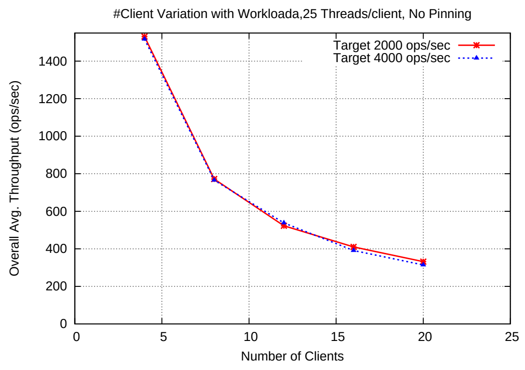

Figure 7: Client variation with different expected throughput.

Figures 4 and 5 show the results of Pinned versus Unpinned set-ups. In the Pinned set-up, through two level pinning, every JVM had access to two physical cores. Figure 3 shows our lab set up with JVM pinning. We tried to design a symmetric set-up to avoid any unnecessary bias in resource allocation for a JVM. 8 JVMs were running on 4 VMs, two on each. Each VM was assigned 2 vcpus. Each vcpu was pinned to 2 physical cores each. Each JVM was pinned to 1 vcpu. *taskset*, a Linux utility that internally uses *sched_setaffinity* was used for pinning, along with changes in the VM configuration file. The number of clients were varied from 4 to 20. Two sets of experiments were conducted: a) every YCSB client is started with 25 threads each, b) the overall thread count in the system from the client's perspective remains fixed. In the former case, with an increase in clients there is an increase in the total number of threads used to create the workload (4 clients, 25 threads each, for a total of 100 threads, 8 clients, 25 threads each, for a total of 200 threads), in the latter irrespective of the number of clients, the total number of threads is fixed at 200. In other words, for the latter, the threads/per client vary to generate the workload based on the number of clients. This thread count refers to the *number of client threads*, a combination of target throughput and number of client threads is used to fine-tune the workload. The overall system level threads in the Hazelcast cluster from the server's perspective remains fixed at all times. We wanted to see if the client thread count has any significant impact on performance. In the load phase, default hashed inserts were used with *Workloada* consisting of 50% gets and 50% puts. In the transaction phase, zipfian distribution was used setting the target to 4000 ops/sec for all the experimental runs. As seen in Figure 4, there is no performance variation with pinning. Even thread count variation does not affect performance indicating no thread migration or contention due to thread restarts on specific cores or context switch overhead. Figure 5 shows an increase in update latency with an increase in clients for an increasing thread count. However, the average update latency perceived by each client does not deviate much if the overall threads used to generate workload remain fixed. The same trend is observed in Figure 6 for the 95th percentile latency. This indicates that when the workload is well distributed across the clients by keeping the overall client thread count constant, there is less variation of average response time. Every client connects to one instance and delegates its requests to that member. Clients are distributed equally among the 8 server instances to avoid bottlenecks. Increasing threads per client further increases parallelization, which increases overall latency. Figure 7 illustrates the fact that when demand is doubled the performance does not degrade further. This indicates that the system is already serving at its full capacity, demanding more may even deteriorate per client throughput.

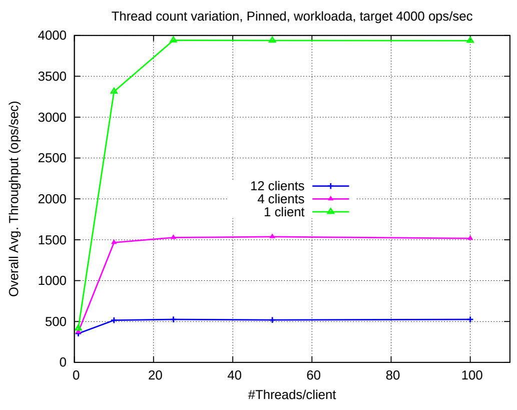

Figure 8: Client side thread variation

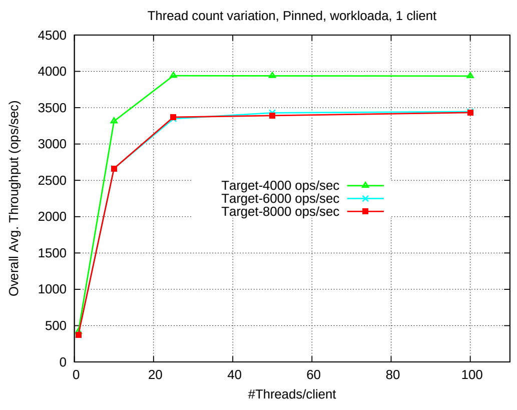

Figure 9: Single client - no contention

Takeaways: Pinning does not help. A well distributed workload across multiple clients improves the perceived response time. We do not consider the overall execution time as our aim is to detect the primary source of contention and its performance implication on the clients. Until the system reaches saturation, increasing demands do not affect throughput much. Beyond the saturation point degradation is expected. 4000 ops/sec is the saturation point in our system setup.

4.3 Thread Count Variation

This section discusses performance behavior with client side thread count variation as mentioned earlier. We get to understand the threshold of the system. Considering a single client with no contention. Figure 9 shows that the system hits saturation at 25 threads beyond which there is no further performance improvement. The system is unable to cope with incoming requests when demand is higher than 4000 ops/sec, after which degradation starts. Figure 8 illustrates that increasing clients degrades per client throughput. The overall system throughput considering all clients from the cluster's perspective never exceeds 6500 ops/sec. When using 25 threads/client, which is our system threshold, the aggregate throughput of all clients hits as high as 6300

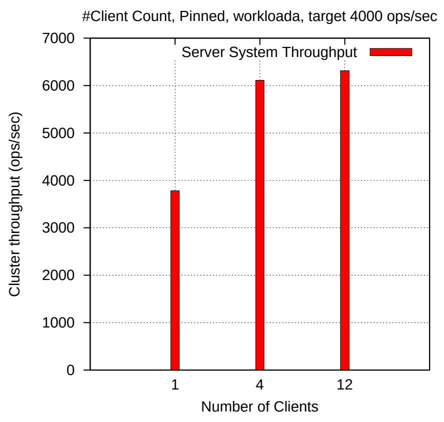

Figure 10: Cluster Throughput.

ops/sec as shown in Figure 10.

Takeaways: Considering the limited scale of our experiments, 25 is the thread count beyond which we do not see any performance improvement. With increasing clients, contention increases and per client throughput decreases. However, increasing the number of client side threads *does not* degrade performance.

4.4 Insert Order and Distribution Type

The YCSB load phase uses hashed inserts by default where keys get hashed to specific slots in the database. In case of ordered inserts, the keys get inserted based on the order of keys. Experiments were conducted with ordered inserts and uniform distribution to see if the way keys are inserted and retrieved impacts performance. Intuitively, it depends on the Hazelcast instance location where keys get stored.

Figure 11 shows that hashed inserts are as good as ordered inserts. We do not see any tangible performance variation whether the transaction phase follows a zipfian or uniform distribution in Figure 12. This seems to imply that if all the clients have a similar transaction pattern, concurrent data access invariably causes some amount of contention no matter where data is stored.

4.5 Workload Variation

Most experiments were conducted with `Workloada` with 50% gets and 50% puts. Two experiments were conducted with other workload types to observe characteristic behavior. While `workloadc` is read only, `workloadb` has 95% reads and 5% updates. `Workloadd` with 95% reads and 5% inserts and `Workloadf` with 50% reads and 50% *read-modify-writes* were also used in one experiment. Figure 13 confirms that pinning has no impact on throughput for different workload types. The larger the fraction of updates, the lower is the throughput. However, *read-modify-writes* in `Workloadf` are costlier than updates. Figure 14 shows that updates take more than twice the time than reads.

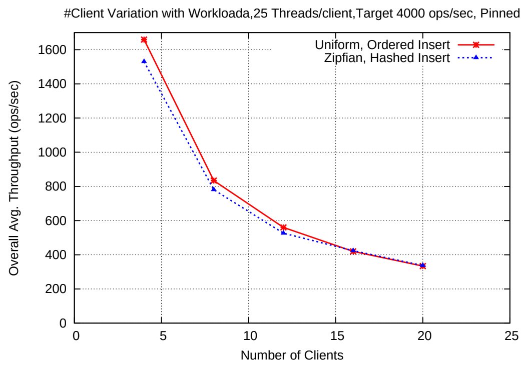

Figure 11: Comparison of Hashed insert & zipfian distribution versus Ordered insert & uniform distribution.

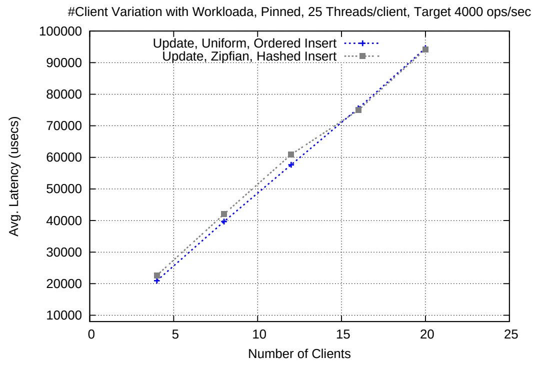

Figure 12: Latency variation with Hashed/Zipfian versus Ordered/Uniform distribution

Takeaways: Prior work (Atikoglu et al. [13]) mentions that reads are more popular in such stores than writes. Since even read throughput is considerably throttled due to contention, improving operational latency is important for better tenant service.

4.6 Multiplexing Client End Points

This section describes our observation with source code changes in Hazelcast. We tailor it to pipeline multiple client requests processed through a single proxy entry point. Our observations indicate a performance improvement with increasing clients. Since prior experiments confirmed that keeping the number of system level threads fixed while increasing client end points invariably degrades throughput, we wanted to see if performance improves by reducing the additional runnable threads in some way by multiplexing multiple TCP connections. This would reduce the additional runnable threads creation with concurrent data structures like lists and queues to store tasks and process them. Our preliminary study indicates a performance improvement over non-multiplexed connections. The `netty` [5] library was used to multiplex multiple TCP connections with changes in Hazelcast's `nio`-based connection implementation. After the socket binds with a specific client port, the contents of the inbound channel is pipelined to the contents of the

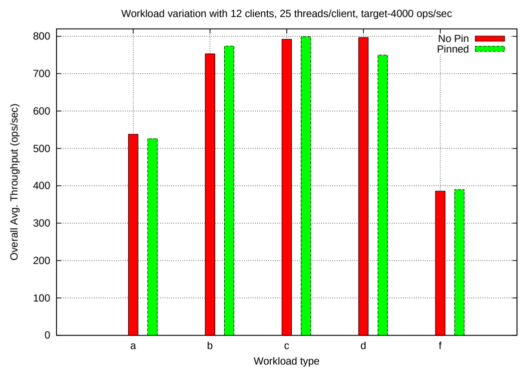

Figure 13: Workload Type Variation.

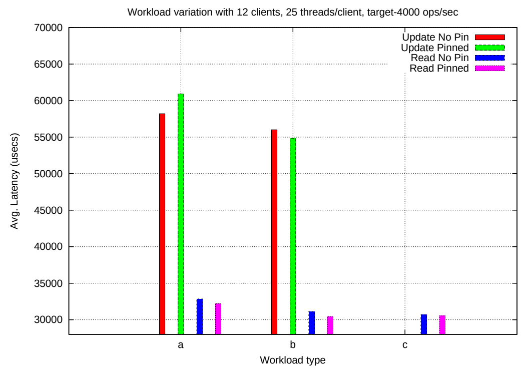

Figure 14: Latency variation with various Workload Types.

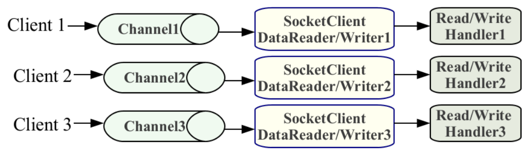

Figure 15: Original Socket Acceptor in Hazelcast.

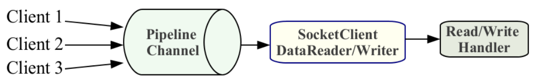

Figure 16: Pipelining Socket channels in Hazelcast.

outbound channel, which is intended to append contents of multiple client data. Figure 15 shows the original Java client protocol in Hazelcast. For every client a separate instance of `SocketClientDataReader/Writer` is created and the handlers are started which reads/writes data to/from socket. Figure 16 shows the modifications made by passing the contents of multiple socket channels pipelined into a single channel before processing it. Each experiment was repeated 3 times and both mean and standard deviation is reported.

Figure 17 shows the improved performance with multiplexed connections with as many as 8 clients. As the number of clients increases pipelining overhead increases and there arises a problem with buffer allocation and writing on the outbound channel. However, a marginal improvement in

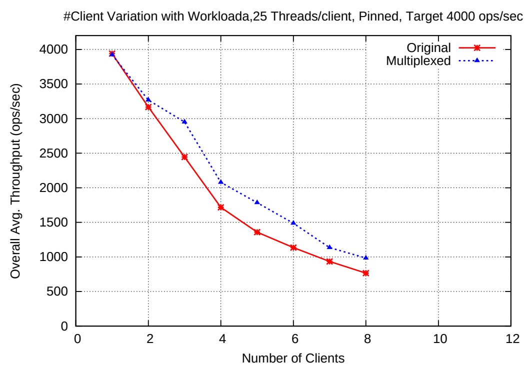

Figure 17: Multiplexed connections.

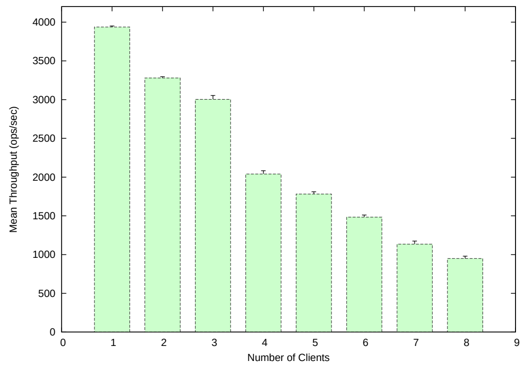

Figure 18: Mean and Standard Deviation of Throughput for multiplexed channels.

throughput justifies our claim that, indeed, multiple threads started for every client right at the outset in an instance causes contention, even though the internal data structures used are *asynchronous non-blocking*. These threads spawning multiple additional threads along the way indeed impact overall performance. Figure 18 shows the mean and standard deviation of the experiments. The standard deviation did not exceed 50 and the percentage increase in throughput is more than 20% for certain number of clients.

5. Result Analysis

To summarize, we conducted experiments on Hazelcast to study performance characteristics with multiple clients. Multiple client end points create new runnable threads with separate data structures on every member, which causes some contention in the cluster. Our goal was to study the following:

- Performance characteristics of Hazelcast with increasing clients under different scenarios.
- Source of degradation of throughput.

We found that pinning JVMs to CPU cores has no impact on performance implying no major overhead of thread migration or context switching between threads. Our experiments tried to answer questions like: How high is the performance degradation? Does workload type and client side thread count have any significant impact on performance? What is the system threshold level when it is at its peak? What causes such degradation? By statistical multiplexing (only for client endpoints, no member-to-member communication) of client TCP connections, we observe a performance improvement. This indicates that newly created threads accepting client data, even though they share resources internally without any explicit blocking, cause overhead resulting in decreasing overall throughput. Multiplexing multiple distinct socket channels and passing them as one proxy on behalf of multiple clients reduces contention.

6. Related Work

A significant amount of work has been done in the context of *multi-tenancy* in distributed shared storage, *performance isolation* and *efficient resource sharing & management*. We discuss here how prior state-of-the-art is relevant in our context and discuss briefly.

- **Centralized Solutions**: *Pisces* [38], Google's *Borg* [43] are centralized schedulers, either intrusive or having a single source of bottleneck preventing good performance with scale. *MROrchestrator* [36] is a resource orchestrator typically for Hadoop dealing with map reduce jobs and is more of a resource allocator than dealing with interference. *Delay scheduling* [47] similarly has worked on HDFS system. How HFS (Hadoop Fair scheduler) alone will work for data-grids is a question.
- **Decentralized Solutions**: *Apollo* [14], *Sparrow* [34], *Omega* [35], although truly decentralized, propose a comprehensive scheduler infrastructure, which is too complex for our targeted systems. Their focus is not on ensuring consistent tenant performance. Apollo aims at highly utilized, load balanced clusters different from our goals. Sparrow proposes a scheduler whose batch sampling and late binding approach is costly and unsuitable for very short task durations like ours. *Mercury* [28] aims at supporting diverse applications through a *hybrid* resource management framework different from our problem goal. Although these refer to resource sharing and fairness, a mechanism maintaining a strong *performance* guarantee in a dynamic environment remains a challenge.

  Although *Cake* [46], *Mesos* [25] and *Yarn* [42], claim to be decentralized, they are either two-level schedulers or have logical centralized scheduling points creating performance bottlenecks. Moreover, they are not fine-grained, focusing on Hadoop-style applications.
- **Cache Replacement Schemes**: *Gdwheel* [29] and *Camp* [24] propose new key replacement strategies to have high cache hits based on not just recency or frequency but also re-computation costs in contrast to our objectives.

- **Cluster Load Balancing**: *MBal* [17], *Scads* [41] and *CloudScale* [37] perform load balancing in a cluster mitigating hotspots through key replication or data migration differing from the context of our work.
- **Coarse Grained Isolation Techniques**: *Argon* [44], *Pulsar* [12], *SQLVM* [32], EyeQ [27], Das et al. [19], *IOFlow* [40], Zeng et al. [48], Walraven et al. [45], *PriDyn* [26] focus on fairness, either at a coarser granularity dealing with virtual machines, network isolation, higher level abstraction or centralized quantum-based scheduling mechanisms in contrast to our work.
- **Workload Insights**: Anderson et al. [11] quantify key-value store consistency through evaluation. Atikoglu et al. [13] unveil workload insights of real-life traces, which can have implications on cache configuration.

Although considerable prior work has looked into performance in distributed storage systems, focusing on sources of contention and ensuring consistent client response remains challenging particularly in key-value stores. The problem of performance degradation is clear, it exists and proposing simple solution to it, even if intrusive, can help everyone using multi-tenant key-value stores.

7. Conclusion and Future Work

In this work we study an in-memory data-grid called Hazelcast in the context of multi-tenancy. Experiments were conducted to see Hazelcast's performance under varied conditions with multiple clients. Our findings unveil insights to the probable cause of contention. New threads for every client end point with its own data structures and spawning other internal I/O threads with a work queue per member instance causes contention. Even though internal data structures used are *asynchronous-nonblocking* and intended for sharing, an increase in end to end instances with increasing clients causes resource contention in such in-memory grids such that degraded overall throughput is significant. Our preliminary results indicate that statistical multiplexing of client socket channels to process requests with fewer runnable instances can alleviate contention.

Although multiplexing TCP channels is not new, it gives a new perspective of looking at the problem. Instead of determining how to partition data or allocate shared resources or schedule requests, the best way to process data through multiple entry points, should be analyzed further. Pipelining channels to avoid excessive parallel threads considering the scale of a cluster with the objective of maximizing performance should be a point of further investigation. In addition to consistent performance we plan to diagnose failures in similar cluster based systems which impacts the overall performance of the storage system.

References

[1] Amazon dynamodb: http://aws.amazon.com/dynamodb/.

[2] Cloudant cloud service: https://cloudant.com/.

[3] Couchdb nosql database: http://couchdb.apache.org/.

[4] http://hazelcast.org/.

[5] http://netty.io/.

[6] Joyent cloud services: https://www.joyent.com/.

[7] Mongodb: http://www.mongodb.org/.

[8] Riak nosql solution: http://docs.basho.com/riak/latest/.

[9] Ycsb: https://github.com/brianfrankcooper/ycsb/wiki/getting-started.

[10] zbase: http://code.zynga.com/2013/08/zbase-a-high-performance-elastic-distributed-key-value-store/.

[11] Eric Anderson, Xiaozhou Li, Mehul Shah, Joseph Tucek, and Jay J Wylie. What consistency does your key-value store actually provide. In *Proceedings of the Sixth international conference on Hot topics in system dependability*, pages 1–16. USENIX Association, 2010.

[12] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg OShea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 233–248. USENIX Association, 2014.

[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[14] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2014.

[15] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[16] 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. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[17] Yue Cheng, Aayush Gupta, and Ali R Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, page 4. ACM, 2015.

[18] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[19] Sudipto Das, Vivek R Narasayya, Feng Li, and Manoj Syamala. Cpu sharing techniques for performance isolation in multi-tenant relational database-as-a-service. *Proceedings of the VLDB Endowment*, 7(1), 2013.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[21] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. *SIGCOMM Comput. Commun. Rev.*, 42(4):25–36, August 2012.

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

[23] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: An active distributed key-value store. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[24] Shahram Ghandeharizadeh, Sandy Irani, Jenny Lam, and Jason Yap. Camp: a cost adaptive multi-queue eviction policy for key-value stores. In *Proceedings of the 15th International Middleware Conference*, pages 289–300. ACM, 2014.

[25] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[26] Nitisha Jain and J Lakshmi. Pridyn: Framework for performance specific qos in cloud storage. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 32–39. IEEE, 2014.

[27] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazieres, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. Eyeq: practical network performance isolation at the edge. *REM*, 1005(A1):A2, 2013.

[28] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters.

[29] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, page 5. ACM, 2015.

[30] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[31] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. Cassandra: An ssd boosted key-value store. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 1162–1167, March 2014.

[32] Vivek R Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR*, 2013.

[33] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *nsdi*, volume 13, pages 385–398, 2013.

[34] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[35] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[36] Bikash Sharma, Ramya Prabhakar, Seung-Hwan Lim, Mahmut T Kandemir, and Chita R Das. Mrorchestrator: A fine-grained resource orchestration framework for mapreduce clusters. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 1–8. IEEE, 2012.

[37] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011.

[38] David Shue, Michael J Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, pages 349–362, 2012.

[39] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[40] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.

[41] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *FAST*, pages 163–176, 2011.

[42] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[43] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[44] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, volume 7, pages 5–5, 2007.

[45] Stefan Walraven, Tanguy Monheim, Eddy Truyen, and Wouter Joosen. Towards performance isolation in multi-tenant saas applications. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, page 6. ACM, 2012.

[46] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: enabling high-level slos on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 14. ACM, 2012.

[47] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[48] Jiaan Zeng and Beth Plale. Multi-tenant fair share in nosql data stores. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 176–184. IEEE, 2014.