Performance Analysis of a Multi-Tenant In-memory Data Grid

Anwesha Das, Frank Mueller, Xiaohui Gu
North Carolina State University
Email: {adas4,fmuelle}@ncsu.edu, gu@csc.ncsu.edu

Arun Iyengar
IBM T. J. Watson Research Center
Email: aruni@us.ibm.com

***Abstract*—Distributed key-value stores have become indispensable for large scale low latency applications. 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 fluctuating workloads that need to scale 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. Through experimental analysis, this paper uncovers scenarios of performance degradation followed by optimized performance via end-point multiplexing. Our study suggests that processing increasing number of client requests spawning *fewer* number of threads help improve performance.**

***Keywords*-Multi-Tenancy; Performance Evaluation; In-memory Data Grid; Key-Value Store.**

I. Introduction

Performance of storage systems and in-memory key-value stores have been the focus of research for a while. 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. MemcacheD [1] in particular has been used by researchers [2], [3] to solve problems related to key-value stores. Besides commercial usage [3]–[10], novel key-value stores have been contributed from academia e.g. Silt [11]. Sustained performance is a primary concern for tenants accessing such stores under fluctuating workloads. In this paper, we present our study of Hazelcast [12], an open source in-memory data grid that supports multi-tenancy. We have conducted experiments to substantiate the fact that performance degradation is indeed high as the number of clients increases. Our study suggests that an increase in the number of *executing threads* with rising number of client end-points is one cause. We further show how to alleviate performance degradation in Hazelcast. We believe that our experimental evidences can help indicate a viable solution to improve performance in similar multi-tenant architectures.

II. Background

With in-memory data grids being used widely, shared cluster storage catering to multiple tenants still suffers from performance problems.

This work was supported in part by NSF grants 1217748 and 0958311.

A. Challenges

Key-value stores cater to a combination of read (get) and write (put) requests. Ensuring enhanced throughput for ever increasing numbers of tenants is challenging because:

- Data placement and eviction in such stores is oblivious to external tenant and data characteristics. E.g. if tenant A's and B's keys share the same cluster instance and every time B's data is accessed A's data gets evicted, then A may experience low throughput.
- Every operation accesses data that was previously stored. Hence, *co-location of data and computation* is important in such clusters. Multiple tenants may need to be serviced by the same instance but, 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 such as workload skews and fluctuating request patterns. A system coping with such fluctuations that tries to deliver the desired throughput encounters resource 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 particularly *overwhelming* for low latency operations.
- Based on workload size and type, throughput tends to vary. Ensuring tenant performance is a challenge because of the inherent, unpredictability of the workload.

B. Hazelcast

Hazelcast [12] is an open source in-memory data grid for distributed computing. Hazelcast's decentralized performance benefits and its easy deployment makes it a good choice for our study. Figure 1 gives an overview of Hazelcast's threading model. Although several parameters like *client-thread-pool* are configurable, every cluster instance by *default* has 7 threads serving I/O operations and 5 threads handling events. Additionally, there are dedicated threads to perform partition aware, generic, or priority operations (see [12] for details). A new client operating in an instance is expensive since several multi-threaded operations are

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.142

IEEE computer society

Authorized licensed use limited to: Stanford University. Downloaded on December 18,2020 at 08:09:32 UTC from IEEE Xplore. Restrictions apply.

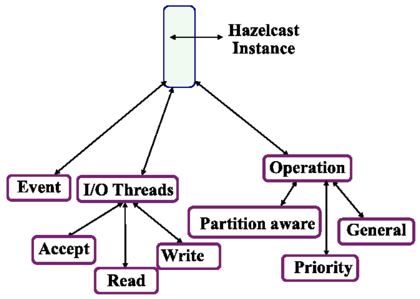

Figure 1. Threading Model

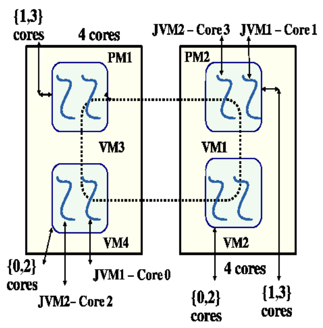

Figure 2. Two-level pinning

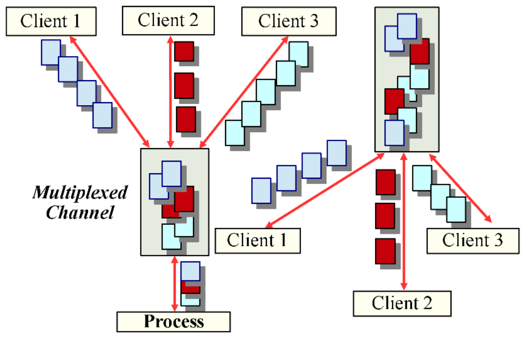

Figure 3. Multiplexing Channels in Hazelcast

associated with a client. Thus, with increasing number of clients per instance, there is an increase in the client threads per instance, and the more 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 increases internal resource consumption through threads and work queues which could create imbalance. Section III-C discusses the performance impact of increasing client connections, threads, and queues.

III. Experimental Evaluation

This section describes the observed performance in terms of overall throughput for varying clients. The YCSB [14] benchmarking tool is used to generate load on the system.

A. *Evaluation Methodology*

Experiments were conducted on a local cluster, where each cluster node is equipped with a quad-core Xeon 2.53GHz CPU and 8GB memory connected to a Gigabit network switch. 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. An 8 instance Hazelcast cluster is setup across 2 hosts and 4 VMs. *JVM* and *Hazelcast server instance* are used interchangeably to indicate a Hazelcast cluster instance henceforth. A separate host outside the cluster ran multiple instances of the YCSB client to create a multi-tenant workload.

B. *JVM-VCPU Pinning*

This section observes the effects of pinning a JVM to a specific physical core. The idea is to prevent thread migration across cores, to increase cache locality, and to reduce overall context switch overhead which might arise due to contention. Figure 2 depicts a two-level pinning, where every JVM has access to two physical cores. 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 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. Two sets of experiments were conducted: a) every YCSB client was started with 25 threads, b) the overall thread count in the system from the client's perspective remained fixed. Clients are distributed equally among the 8 server instances to avoid unevenness. 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 number of system-level threads created by the Hazelcast cluster remains fixed at all times from the server's perspective. We wanted to see if the client thread count had any significant impact on performance. In the load phase, `Workloada` was used consisting of 50% gets and 50% puts. In the transaction phase, a *Zipfian* distribution was used setting the *target* (expected per client throughput) to 4000 ops/sec.

As seen in Figure 4, there is no change in performance with pinning. Even a thread count variation does not affect performance. Hence, thread migration context switches do not contribute *significantly* to overhead. Figure 5 shows an increase in update latency with an increase in number of clients with a *larger* thread count. However, the average update latency perceived by each client does not deviate much if the overall number of threads generated by the workload remains fixed. The same trend is observed in Figure 6 for the 95th percentile latency illustrating the fact that 95% of the operations completed under the indicated latency on Y-Axis. 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. Increasing the number of threads per client further increases parallelization, which increases overall latency.

C. *Multiplexing Client Channels*

This section describes our observation with modifications to Hazelcast. We studied and then modified less than 10% of the *Java* source code in an effort to pipeline multiple client requests through a single proxy entry point. Our observations indicate a performance improvement with increasing number of clients. Since prior experiments confirmed that keeping the number of system-level threads created by the Hazelcast

Authorized licensed use limited to: Stanford University. Downloaded on December 18,2020 at 08:09:32 UTC from IEEE Xplore. Restrictions apply.

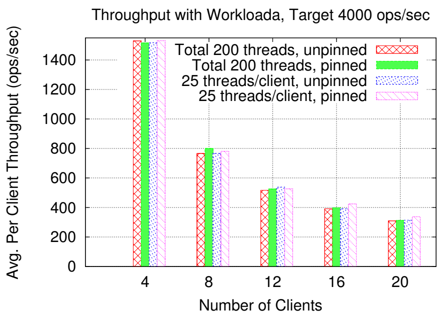

Figure 4. JVM-VCPU Pinned vs. Unpinned

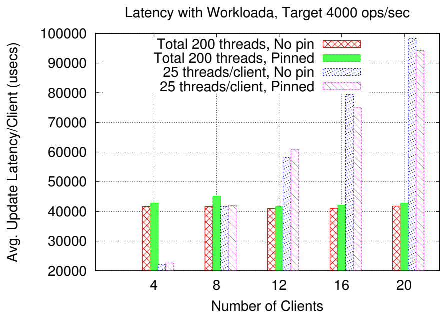

Figure 5. Update Latency Pinned vs. Unpinned

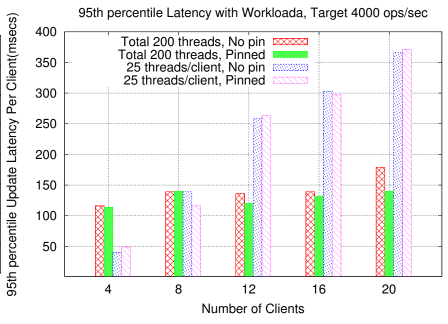

Figure 6. 95th percentile latency plot

cluster fixed while increasing client end points invariably degrades throughput, we wanted to see if performance is affected by reducing the number of threads through multiplexing multiple client connections. This reduces the number of runnable I/O threads needed to handle the client threads. Our preliminary study indicates a performance improvement over non-multiplexed connections. The `netty` [15] library was used to multiplex multiple client connections with changes in Hazelcast's `nio`-based connection implementation. The contents of multiple registered clients (inbound channels) are pipelined to the contents of a single outbound channel, containing multiple client data. This outbound channel talking to the selector reduce the overall channel I/O improving balance per instance. We noticed that hazelcast created lesser number of threads (I/O, service, execution) than the naive implementation owing to the reduced outbound channel. Figure 3 shows the modifications made by passing the contents of multiple socket channels into a single channel before processing it. Each experiment was repeated 3 times and both mean and standard deviation are reported. Figure 7 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 *per client throughput* justifies our claim that, indeed, multiple threads started for every client right at the outset causes contention, even though the internal data structures used are *asynchronous and non-blocking*. These threads, spawning multiple additional threads along the way, impact overall performance. Figure 8 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 15% for certain number of clients (see Figure 7).

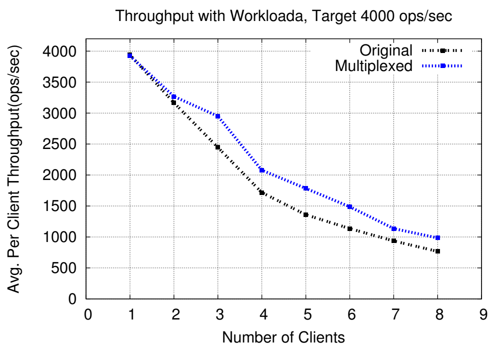

Figure 7. Multiplexed connections

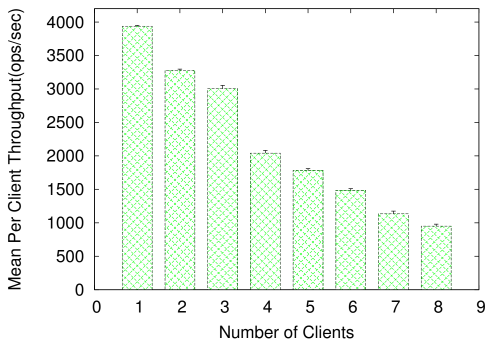

Figure 8. Mean and Standard Deviation of Throughput

IV. Limitations

Some experiments were conducted by over-stressing the system over limited scale. This is a limitation considering the fact that demanding much higher than the system's maximum threshold (which is likely in a real-life scenario) is not the correct way to perform experimental evaluations. Our results are limited to 8 clients, only. More research needs to be conducted into statistical multiplexing of diverse client requests to a cumulative outbound channel so as to prevent buffer overflow. With higher numbers of clients & limited buffer sizes, mapping n client requests through m channels needs to be further investigated. Moreover, multiplexing approach is intrusive and may be inconvenient for deployment in stores having message-based communication (Cassandra [4] uses RPC-Remote Procedure Calls). However, the focus in this paper has been fast in-memory key-value stores. These applications mostly use at least one if not all of the Java/ReST/MemcacheD-style client protocols, which internally use TCP/HTTP. Hence, our idea and insights to performance improvement can be easily

Authorized licensed use limited to: Stanford University. Downloaded on December 18,2020 at 08:09:32 UTC from IEEE Xplore. Restrictions apply.

extrapolated to the majority of such popular stores.

V. Related Work

We discuss briefly how prior state-of-the-art is relevant in our context. Centralized Solutions: *Pisces* [2], Google's *Borg* [16] are intrusive centralized schedulers focusing on resource allocation and management complementing our work focusing on *contention analysis*.

Decentralized Solutions: *Apollo* [17], *Sparrow* [18], *Omega* [19], *Mercury* [20], *Cake* [21], *Mesos* [22], and *Yarn* [23] either propose a comprehensive scheduler infrastructure, which is too complex and unfit for our target systems or are not fine-grained enough focusing on Hadoop-style applications.

Although prior work has investigated performance of distributed storage systems, focusing on sources of contention and ensuring consistent client response remains challenging. The existence of performance degradation was demonstrated (Figure 4) and a solution (Section III-C), even if intrusive, was shown to especially aid cloud computing infrastructures.

VI. Conclusion

In this paper, we study an in-memory data-grid called Hazelcast in the context of multi-tenancy to assess its performance. Our findings unveil the following interesting insights: 1) JVM pinning does not help; 2) maintaining several threads per client with its own data structures and spawning other internal I/O threads with a work queue per member instance degrades throughput. Even though internal data structures used are *asynchronous and nonblocking* intended for sharing, an increase in end-to-end instances with increasing clients causes resource contention. Our preliminary results indicate that multiple client requests with *fewer runnable instances* can alleviate contention. This analytical study provides 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.

References

[1] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, Aug. 2004.

[2] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage." in *OSDI*, 2012, pp. 349–362.

[3] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook." in *NSDI*, vol. 13, 2013, pp. 385–398.

[4] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen, "Cassandra: An ssd boosted key-value store," in *International Conference on Data Engineering*, March 2014, pp. 1162–1167.

[5] J. L. Carlson, *Redis in Action*. Greenwich, CT, USA: Manning Publications Co., 2013.

[6] "Mongodb: http://www.mongodb.org/."

[7] "Amazon dynamodb: http://aws.amazon.com/dynamodb/."

[8] "Cloudant cloud service: https://cloudant.com/."

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

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

[11] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Symposium on Operating Systems Principles*, 2011, pp. 1–13.

[12] "http://hazelcast.org/."

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, 2012, pp. 53–64.

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

[15] "http://netty.io/."

[16] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *European Conference on Computer Systems*, 2015, p. 18.

[17] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *Operating Systems Design and Implementation*, 2014.

[18] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Symposium on Operating Systems Principles*, 2013, pp. 69–84.

[19] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *European Conference on Computer Systems*, 2013, pp. 351–364.

[20] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," pp. 485–497, 2015.

[21] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: enabling high-level slos on shared storage systems," in *ACM Symposium on Cloud Computing*, 2012, p. 14.

[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, 2011, pp. 22–22.

[23] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Symposium on Cloud Computing*, 2013, p. 5.

Authorized licensed use limited to: Stanford University. Downloaded on December 18,2020 at 08:09:32 UTC from IEEE Xplore. Restrictions apply.