Distributed shared memory through key-value stores

Mantas Serapinas

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2018

Abstract

* Still to be written.

Acknowledgements

* Still to be written.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Distributed shared memory (DSM) is memory architecture where physically distributed memory can be accessed as one logically shared address space. Systems based on shared memory architecture reduce the complexity of parallel programming (Z.Huang et al., 2006). Unfortunately, building an efficient distributed shared memory system is a huge challenge and the documentation on the existing open-source DSMs is rather limited. Thus it can be a daunting task to run parallel programs on distributed shared memory systems.

However, with cloud computing becoming increasingly popular new solution became available, namely NoSQL data stores (Grolinger et al., 2013). NoSQL can be completely schema-free, most popular data models being key-value stores, document stores, column-family stores, and graph databases. It is able to scale horizontally over many commodity servers. On top of that, some cloud data management systems provide strong consistency model, which means that after update operations all nodes agree on the new value before making it available to the user. All these properties make it possible to use such data stores as distributed shared memory.

The focus of this project is to expose the distributed shared memory model in a cloud by implementing an instrumentation tool which translates load and store instructions to get and put calls to key-value store. This tool will let users to run parallel programs on cloud using key-value store without editing a single line of code.

1.2 Scope

The initial goal of the project was to create the tool which can instrument programs written in any user preferred language but due to communication with Google data store (using gRPC and protocol buffers) a separate gRPC library for each language is

needed. The task is trivial but in order to meet the project deadline a single language was chosen, namely C++.

1.3 Approach

The first phase of the project was about choosing an appropriate key-value store. Google data stores were selected for further benchmarking as they are well documented and widely used in the industry. As Bigtable both showed the best results in throughput and latency, and provides strong consistency, it was chosen to be used as the representative data store for the project.

The translation of load and store instructions were implemented by writing an LLVM pass. The pass iterates over the instructions and changes load and store instructions to get and put function calls to data store, respectively. Moreover, a new malloc function was created in order to preserve the user program from allocating heap memory for objects, which are stored in Bigtable.

1.4 Contributions

The contributions of this paper are as follows:

- Research was done on Google data stores, namely Bigtable, Datastore and Spanner, their features and the consistency models they provide.
- Benchmarking the above data stores based on their throughput and latency using YCSB tool.
- Research on possible ways to translate load and store instructions (Intel PIN, LLVM).
- Implementing an LLVM pass to do the translation and linking it with gRPC, protobuf and googleapis libraries to communicate with Bigtable instance table.
- Implementing a malloc() function to preserve the user program from allocating heap memory for objects.

1.5 Synopsis

Chapter 2 presents the main requirements for the data stores to be used as distributed shared memory systems. The chapter continues with the background information on the selected Google Cloud data stores, namely Bigtable, Datastore and Spanner. Finally, the chapter discusses the results of the benchmark ran on these data stores.

Chapter 3 starts with the architecture of the tool, also briefly introducing gRPC and protobuf libraries. Then, the chapter briefly talks about the unsuccessful attempt to

translate store and load instructions to get and put operations on data store using Intel PIN tool. The chapter continues with an LLVM pass implementation.

Chapter 4 presents the memory wasting problem, introduced by storing heap variables on Bigtable, and describes the solution - the implementation of custom heap memory allocation functions.

Chapter 5 discusses the correctness and efficiency of the system.

Chapter 6 introduces the API which lets computers on two different locations in the world use the key-value store as distributed shared memory in scenarios like producer/consumer.

Chapter 7 summarizes the work done and possible ways of improving the system.

Chapter 2

Data store for DSM

2.1 Overview

In order to build and test the translation tool, a single cloud data store was chosen to be used as a distributed shared memory system for the project. The main requirements for the data store were:

- provide efficient throughput and latency results;
- provide strong consistency model;
- have a way to run user programs on the same data centre, the data store is located on;
- provide an API to communicate in C++;
- preferably provide key-value database model.

Three Google cloud storages, which met almost all of the requirements, were suggested, namely Bigtable, Datastore and Spanner. Even though neither of the three candidates had key-value store as their primary database model, they were one of the few that provided communication between C++ program and a data store. Google cloud products provide this functionality through gRPC (open source remote procedure call system) using protobuf library and Google APIs. Moreover, all of these Google data storages can be chosen to be located in the same data centre for best throughput and latency results. Further sections provide a brief look into each of the candidates and show the results of the benchmarking on throughput and latency.

2.2 Bigtable

Bigtable (Google, 2018a) is high performance, wide column NoSQL database, which stores data in massively scalable tables, each of which is a sorted key/value map. Tables

consists of rows, each of which is essentially a collection of key/value entries, where the key is a combination of the column family, column qualifier and timestamp.

Bigtable treats all data as raw byte strings. If a row does not include a value for a specific key, the key/value pair simply does not exist. Changes to a row take up extra storage space, as Bigtable stores mutations sequentially and compacts them only periodically, but as the usual amount of data sent from our tool does not exceed 32/64 bits (depending on the machine architecture) the additional amount of memory used is insignificant.

Most importantly, Bigtable supports look up value associated with key operation and provides strong consistency - all writes are seen in the same order.

2.3 Datastore

Datastore (Google, 2018b) is highly-scalable NoSQL, document store model database developed by Google. Unlike Bigtable, it provides a SQL-like query language (GQL) and ACID (Atomicity, Consistency, Isolation, Durability) properties for atomic transactions. Moreover, it supports a variety of data types, including integers, floating-point numbers and many more, although such functionality is not needed for purpose of the project as the tool stores binary data directly. Datastore uses synchronous replication, meaning that data is written to primary storage and the replica simultaneously.

Similarly to Bigtable, Datastore provides strong consistency for entity (row) lookups by key. It also provides strong consistency for ancestor queries but they are not relevant to the project.

2.4 Spanner

Spanner (Google, 2018c) is a horizontally scalable, globally consistent relational database service. Unlike the previously discussed storages, Spanner has an key-value store as additional database model, data scheme and uses SQL. Similarly to the Datastore, it provides ACID transaction properties.

Spanner provides even stronger consistency property than strong consistency, namely external consistency. External consistency guarantees that for any two transactions, T_1 and T_2: if T_2 starts to commit after T_1 finishes committing, then the timestamp for T_2 is greater than the timestamp for T_1 .

2.5 Benchmarking results

For the benchmarking an existing industry tool was used - Yahoo! Cloud Serving Benchmark (YCSB) (Cooper et al., 2010). A key feature of YCSB, as described by its

developers, is that it is extensible. YCSB is open-source, supports easy definitions of new systems and workloads. Workloads allow to understand the performance tradeoffs of different systems.

The main operations done by the translation tool are reads and writes with a small amount of read-modify-write operations on heap allocation pointer, keeping track of the address to next free memory space. Thus, workloads A and F were chosen, simulating update heavy and read-modify-write using systems, respectively.

For the best results the benchmarking was run on Google Compute Engine (GCE) virtual machine situated at the same data centre as the data stores.

2.5.1 Loading the data

Before running the benchmark on workloads, 1000 rows were inserted into each data store. Figures 2.1 and 2.2 show the latency and throughput achieved by each cloud storage. The results show that both Bigtable and Spanner have much lower latency and higher throughput than Datastore. This can be explained by research results on Datastore using synchronous replication, which makes the host wait until all replications are created, as described in Margaret's Rouse article Synchronous replication (Rouse, 2016).

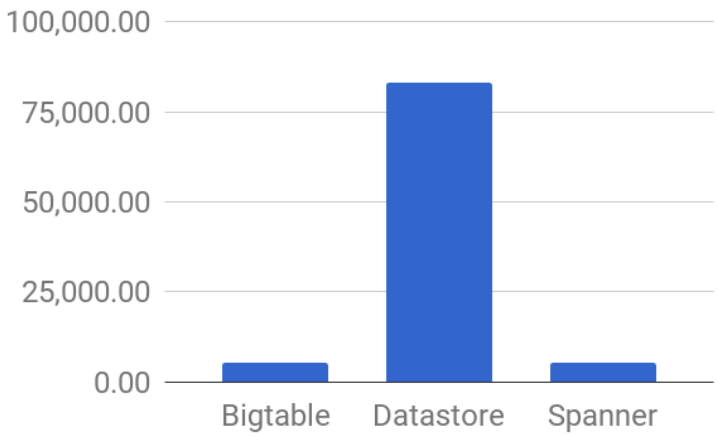

Figure 2.1: Latency (μs) for 1000 insert (write) operations

2.5.2 Workloads

Workload A consists of 1000 operations (500 reads and 500 writes) while workload F consists of 2000 operations (1000 reads, 500 atomic read-modify-write operations and 500 writes). The results of the benchmark in terms of latency on write operations were consistent with the previous loading benchmark results, with Bigtable and Spanner performing significantly better over Datastore (Figure 2.3). The latency on write operations showed a clear dominance by Bigtable.

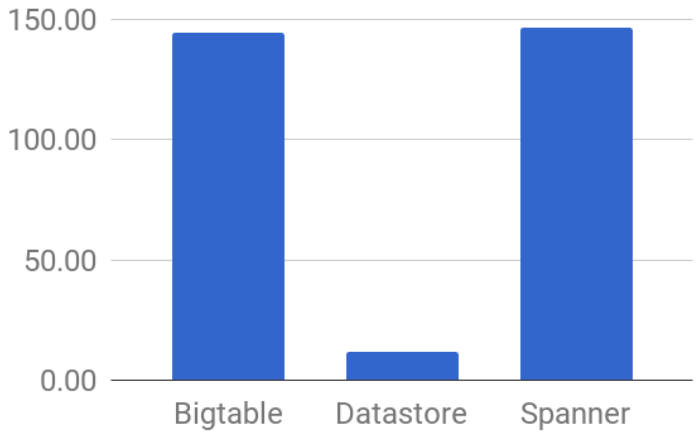

Figure 2.2: Throughput (ops/sec) for 1000 insert (write) operations

Even though, the difference on read operations latency between Datastore and two other data storages were smaller than with write operations (Figure 2.4), Datastore still was more than two times slower than Spanner and more than 4 times slower than Bigtable. The latency results on read-modify-write operations showed a similar trend as read and write operations (Figure 2.5).

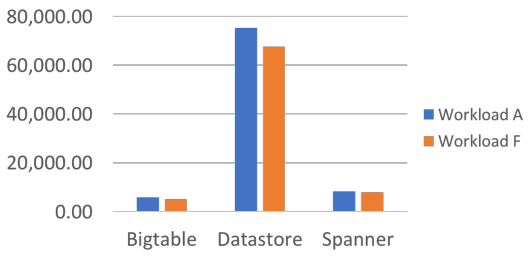

Figure 2.3

The overall throughput, again, showed a significant superiority by Bigtable, as indicated in Figure 2.6.

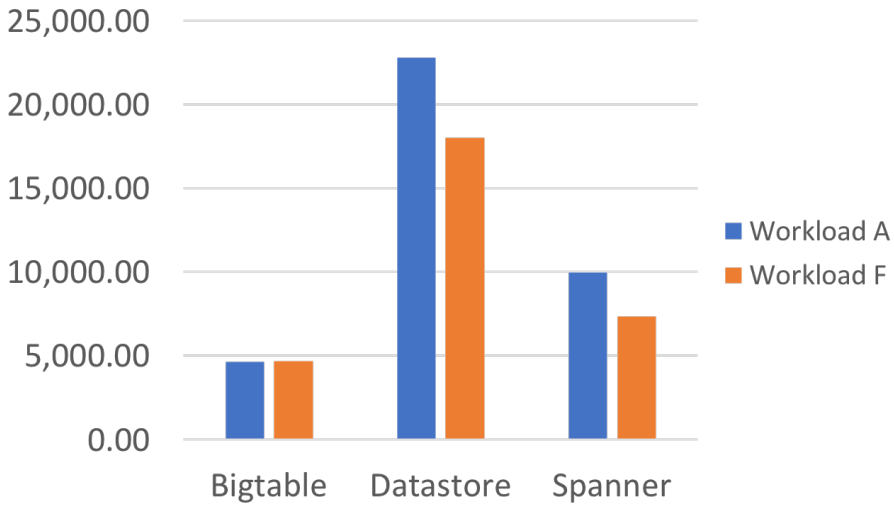

Figure 2.4: Read operations latency (μs) for Workload A (500 reads) and Workload B (1000 reads)

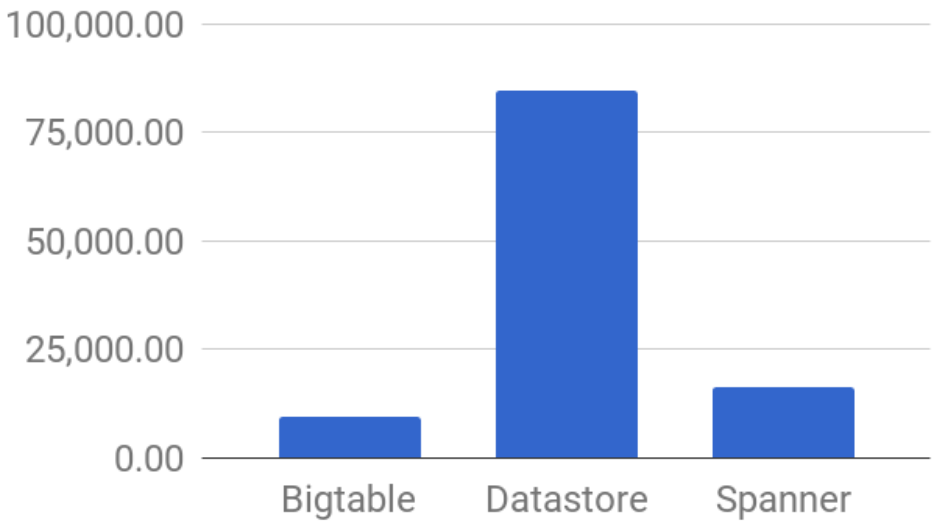

Figure 2.5: Read-modify-operations latency (μs)

Figure 2.6: Throughput (ops/sec) for Workloads A (1000 operations) and Workload B (2000 operations)

2.5.3 Conclusions

As Bigtable showed the best results in loading of data and on both of the workloads the benchmarks were run on, and since it provided a strong consistency model, it was selected to be used as a distributed shared memory system for the translation tool.

Chapter 3

Interim report

3.1 Accomplishments so far

- Research was done on Google data stores. Benchmarked the above data stores based on their throughput and latency using YCSB tool.
- Research was done on OpenSHMEM and POSIX threads and their suitability for the project (hardware/software required).
- Research was done on possible ways to translate load and store instructions (Intel PIN, LLVM).
- Implemented an LLVM pass to call an external functions (dummy get/put) as the Proof of Concept (PoC) application. Found out how to link gRPC, protobuf and googleapis libraries to create external get/put functions. Implemented the get/put functions for simple static and stack variables storage on Bigtable.

3.2 Remains to be done

- Create get/put functions to store heap variables, pointers, pointers-to-pointers, etc. Test it!
- Heap allocation functions implementation and LLVM pass modification to change the calls from the original heap allocation functions (malloc, calloc, etc) to calls to the implemented ones. Test it!
- Create an API which would let imitate the producer-consumer relationship through DSM, with producer and consumer situated in geographically different locations. Bigtable multi regional locations could be used (https://cloud.google.com/storage/docs/bucket-locations#location-mr).

Other possible work is to be determined with project supervisor.

3.3 Timeline for final semester

Task	Deadline
Create get/put functions to store heap variables, pointers, etc.	1 Feb
Implementation of custom heap allocation functions and related subtasks	10 Feb
Create an API	18 Feb
Other tasks (TBD)	4 Mar
Report writing	5 April

Chapter 4

Load and store instructions translation

4.1 Architecture

4.1.1 Overview

4.1.2 gRPC and protobuf

4.1.3 get() and put() functions

4.1.4 Memory management

4.2 Try at Intel PIN

4.3 LLVM pass

4.3.1 Overview

4.3.2 Translation

Chapter 5

Problems with heap allocation

5.1 Overview

5.2 Implementation of malloc()

Chapter 6

Results

Chapter 7

API

Chapter 8

Conclusions

8.0.1 Overview

8.0.2 Future work

Bibliography

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. URL https://www2.cs.duke.edu/courses/fall13/cps296.4/838-CloudPapers/ycsb.pdf.

Google, 2018a. URL https://cloud.google.com/bigtable/.

Google, 2018b. URL https://cloud.google.com/datastore/.

Google, 2018c. URL https://cloud.google.com/spanner/.

Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, December 2013.

Margaret Rouse. Synchronous replication. *WhatIs.com*, 2016.

Z.Huang, W.Chen, M.Purvis, and W.Zheng. Vodca: View-oriented, distributed, cluster-based approach to parallel computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 2, pages 15–15, May 2006.