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

This is an example of `infthesis` style. The file `skeleton.tex` generates this document and can be used to get a "skeleton" for your thesis. The abstract should summarise your report and fit in the space on the first page. You may, of course, use any other software to write your report, as long as you follow the same style. That means: producing a title page as given here, and including a table of contents and bibliography.

Acknowledgements

Acknowledgements go here.

Table of Contents

Chapter 1

Introduction

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 **??**. 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't be a daunting task to get to run ones code on such a system.

However, with cloud computing becoming increasingly popular new solution became available, namely NoSQL data stores **??**. 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.

1.1 Motivation

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) 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 to choose 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, which iterates over the instructions and changes these 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 will be stored in Bigtable.

1.4 Contributions

The contributions of this paper are as follows:

- Research was done on how Google data stores, namely Bigtable, Datastore and Spanner, work and what consistency models they provide.
- Benchmarking the above data stores based on their throughput and latency using YCSB tool.
- Research on OpenSHMEM and POSIX threads and their suitability for the project (hardware/software required).
- 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.

Chapter 2

Choosing data store

2.1 Overview

The first step in the project was to decide on which cloud data store to expose as distributed shared memory system. 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 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++ process and a data store. Google cloud products provides 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, show the results of the benchmarking on throughput and latency.

2.2 Bigtable

Bigtable 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 architecture used) the additional amount of memory used is insignificant.

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

2.3 Datastore

Datastore 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, binary data 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 data is written to primary storage and the replica simultaneously.

On the contrary to Bigtable, Datastore provides strong consistency only for entity (row) lookups by key and ancestor queries, which are not relevant to our needs. The writes to Datastore are only eventually consistent.

2.4 Spanner

Spanner 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. Same as 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 - YCSB. It provides multiple workloads to benchmark the different cloud storage solutions on. The main operations done by the translation tool are reads and writes (with some amount of read-modify-write operations on malloc() pointer keeping track of the next address to next free

memory space). Thus, workloads A and F **??** were chosen, simulating update heavy system 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. The figures **??** - **??** shows the latency and throughput achieved by each cloud storage. The results clearly show that both Bigtable and Spanner has approximately 16 times lower latency and approximately 12 times higher throughput. 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 **??**.

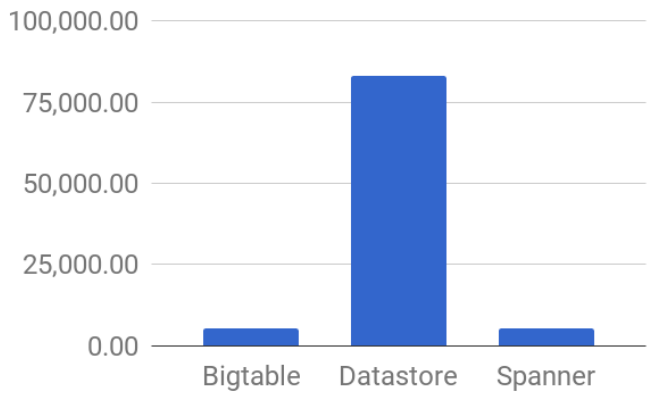

Figure 2.1: Latency (μs) for 1000 insert (write) operations

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 operation were consistent with the previous loading benchmark results, with Bigtable and Spanner performing significantly better **??**. The latency on write operations showed a clear dominance by Bigtable **??**, having two times lower latency. Even though, the difference on read operations latency between Datastore and two other data storages were smaller than with write operations, 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 **??**. The overall throughput, again, showed a significant superiority by Bigtable, as indicated in **??**.

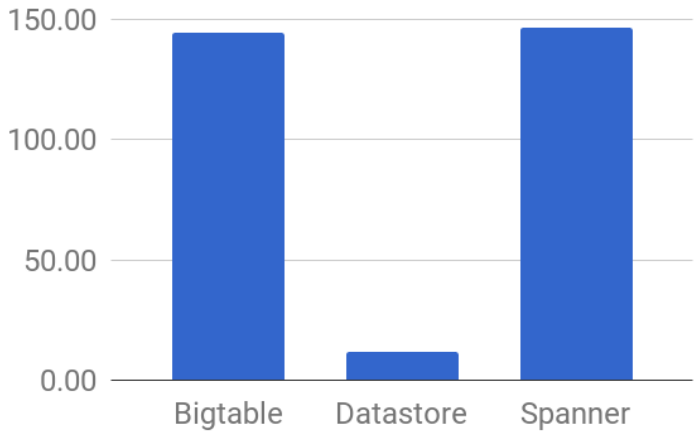

Figure 2.2: Throughput (ops/sec) for 1000 insert (write) operations

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

OpenSHMEM vs POSIX threads

3.1 Overview

3.2 OpenSHMEM API

3.3 POSIX threads

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

4.3.3 Malloc() function

4.3.4 Results

Chapter 5

API?

Chapter 6

Conclusions

6.0.1 Overview

6.0.2 Future work

Bibliography

[1] Hiroki Arimura. Learning acyclic first-order horn sentences from entailment. In *Proc. of the 8th Intl. Conf. on Algorithmic Learning Theory, ALT '97*, pages 432–445, 1997.

[2] Chen-Chung Chang and H. Jerome Keisler. *Model Theory*. North-Holland, third edition, 1990.