Memory through key-value stores

Mantas Serapinas

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

2018

Abstract

Sometimes people come across programs, which cannot fit into main memory. This project proposes a software solution aimed to run a user program using key-value cloud storage service, which provides huge amounts of memory, as the main memory source. The implemented translation tool changes all store and load instructions to put and get operations on Google Bigtable storage using LLVM transformation pass with its `opt` tool. After noticing that virtual memory was still being allocated on heap allocation function calls, two custom heap allocators were implemented: first-fit free-list and the one without freed memory reusage. The benchmarking results showed that the allocator without freed memory reusage works significantly faster than both first-fit free-list and the default allocators. The implementation of a custom heap allocator was advantageous as it made it possible to create a heap memory translator, which only translates store and load instructions which operate on heap memory. The programs instrumented by heap translation tool work faster, but have an huge overhead on the instruction count. Overall, the translated programs work orders of magnitude times slower but some solutions to alleviate this problem were identified.

Acknowledgements

I would like to express my gratitude to my supervisor Vijay Nagarajan for aspiring guidance and advice.

I would also like to thank Harry Wagstaff and Aaron Smith for invaluable advice.

Finally, to my family and friends for their support and encouragement.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Many computing applications demand a large amount of memory space. All this memory has to be provided by the primary memory, also called random access memory (RAM). On some operating systems (i.e. Linux or Mac OS), when the primary memory fills up and more memory is needed for a process, the operating system moves inactive pages in memory to a swap space. Swap space is usually a limited amount of memory located on the secondary storage, thus it provides orders of magnitude slower access times than RAM. The problem arises when even using swap space, the amount of memory is not enough to satisfy memory requirements of a program. Some systems have hard limits on the amount of memory a process can operate on and thus denies any excessive amounts of memory allocations. Other systems behave more wildly, allows to allocate more memory than the system has and eventually starts killing processes by some predefined heuristic (goldilocks, 2016). Neither of the strategies solve the problem of main memory shortage and the only possible solution seems to be physically adding more RAM to the system. This project proposes a software solution aimed to run a user program using key-value cloud storage as the main memory source.

The decision to use key-value cloud storage was based on the progress cloud services made during the last decade. We are particularly interested in NoSQL data stores (Grolinger et al., 2013). NoSQL can be completely schema-free, with 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, thus providing huge amounts of memory storage. Some cloud storage services (i.e. Google Bigtable) even provide in-memory tables, which greatly decreases read and write times in such systems. On top of that, some cloud data management systems provide strong consistency model, which means that after an update operation all nodes agree on the new value before making it available to the user. All these properties let cloud storages be used as the main memory source.

The proposed solution in this project is somewhat similar to the idea of network RAM. When using network RAM, the system pages to idle memory over the network rather

than to disk. This seems like a valid solution from the performance perspective, as the latency of some networks is much lower than the latency of disk seek (Dean, 2009). Although the main purposes of using such systems are described to be performance improvement (Anderson and Neefe, 1998) and better resource utilisation in distributed systems (Oleszkiewicz et al., 2004), it can also benefit as an external memory provider. For our project, the cloud storage stands as an abstraction of parallel network RAM. However, as the concept of a network RAM was discovered in one of the final stages of project development, the proposed solution uses the network RAM (or the cloud storage) as substitution not only for disk but also for main memory accesses. This, of course, makes the instrumented programs very inefficient and is identified as one of the key improvements for future work.

The solutions on how to expose cloud storage as the main memory source, their advantages and disadvantages were identified:

- **Explicit management by user.** This solution requires the user to explicitly move data back and forth from either disk or cloud storage using an API to store and retrieve values from cloud storage. It requires a lot of code modification, thus might not be well accepted by users.
- **Compiler**. This solution compiles the user code with an implemented compiler. The compiler would use cloud storage API to get and put values instead of using store and load instructions. The solution doesn't require any code modification from. However, this solution is not portable with different languages and seems like too excessive and unrealistic given the timespan of the project.
- **Program transformation (LLVM)**. This solution involves compiling the user source code to LLVM IR and passing it through an implemented transformation pass, which translates all store and load instructions to get and put operations on the chosen cloud storage. This solution is more portable as it works will all languages, which have LLVM frontend, and all architectures supported by LLVM.
- **Binary translation**. This solution involves translating machine code from one instruction set architecture to another and is best used when source code is not available. Binary translation is further split into:

  Static. Static binary translation transforms the machine code to work on the target architecture without having to run the code. This approach is more efficient than dynamic binary translation. However, it is not a complete solution, due to problems such as dynamic linking and self-modifying code.

  Dynamic. Dynamic binary translation transforms the code on the fly by looking at the upcoming instructions, translating and caching them any further reference. The translation during runtime incurs a significant overhead but can be amortised when translated instructions are executed multiple times.

After careful examination it was decided to do a program transformation solution using LLVM tools and compiler, as this is one of the most portable solutions, incurs only a minimal overhead during program's runtime (due to increased instruction count) and

does not require any modification to the source code, as the transformation is done during the compilation. Moreover, it is assumed that the source code of the program exists, thus eliminating the need of binary translation.

1.2 Scope

It was understood that an industry quality tool should be as portable as possible and work with any cloud storage. However, due to the limited timespan of the project, it was decided to create a modular proof of concept tool that works with Google Bigtable and could be easily extended to work with other cloud storages. For the same reason, the implemented tool works only with programs written in C and C++. These specific languages were selected as they are both fully supported by LLVM through Clang compiler, while other languages are mainly supported by LLVM community created compiler frontends.

1.3 Contributions

The contributions of this paper are as follows:

- Research and benchmark of multiple Google data stores.
- Implementation of the tool which translates all program's load and store instructions to get and put calls on Bigtable.
- Implemented custom heap allocator.
- Implemented a variant of the translation tool which instruments only the load and store instructions originating from dynamic (heap) memory.

1.4 Synopsis

Chapter 2 presents the main requirements for the data stores to be used as the main memory source. The chapter continues with the discussion about the results of the benchmark ran on three Google cloud data-serving systems, namely Bigtable, Datastore and Spanner, followed by a more in-depth look at the chosen system, Bigtable. Finally, the chapter introduces the three functions implemented to read and write the contents of Bigtable.

Chapter 3 introduces LLVM project, gives a higher level picture of the translation tool followed by a more in-depth look at the implementation of the tool.

Chapter 4 presents the virtual memory waste problem, introduced by storing heap variables on Bigtable, and describes the solution - custom heap memory allocator. Two

heap allocators are introduced: first-fit free-list and the allocator without freed memory reusage. Finally, the chapter discusses the consistency of the proposed allocators on multithreaded applications and finishes with virtual memory usage and performance analysis.

Chapter 5 introduces the implemented heap memory translation tool, which only translates load and store instructions operating on heap memory space.

Chapter 6 presents the evaluation of the two translation tools based on the performance and instruction count of the programs.

Chapter 7 summarizes the work done and proposes a few possible improvements for future work.

Chapter 2

Data store

2.1 Overview

The proposed translation tool instruments the program and makes it use the cloud data store as the main memory source. For the purpose of the project, a single cloud storage solution was chosen. The main requirements for the data store were:

- provide high throughput and low latency read and write operations;
- provide strong consistency model;
- provide a geographical locality for the machine that runs the instrumented program and the data store. In other words, have an option to choose that both machines would be located in the same data centre. This provides better throughput and latency results, as the nodes in the same data centre have very high bandwidth (1 Gbps - 100 Gbps) connections;
- provide a C++ API for data store operations;
- preferably provide key-value database model.

After a brief research three Google cloud storages were selected for further examination: Bigtable, Datastore and Spanner. Even though neither of the three candidates have key-value store as their primary database model, they are one of the few that provide a C++ API for data store operations. This is achieved by using a combination of gRPC and protobuf libraries and Google APIs. Furthermore, all of the three storages have an option to be located on the same data centre as the machine running the instrumented program. All of the three data stores provide strong consistency models, with Spanner providing even stronger model, namely external consistency[1]. Finally, the three selected Google data-serving systems were further analysed by performing a benchmark for their throughput and latency results.

[1]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.

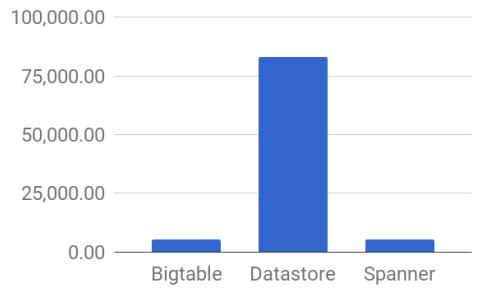
2.2 Benchmarking results

The benchmarking was done using Yahoo! Cloud Serving Benchmark(YCSB) (Cooper et al., 2010) tool. YCSB is an open-source framework for evaluating and comparing the performance of multiple types of data-serving systems. A key feature of YCSB, as described by its developers and users, is its extensibility, as it supports easy definitions of new systems and workloads. Workloads define the data that is loaded into the data store during the loading phase, and the operations are executed against the data set during the transaction phase. Workloads allow to better understand the performance tradeoffs of different data-serving systems.

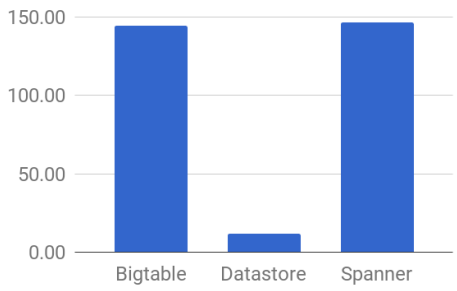
The main operations done by the program instrumented with the translation tool are reads and writes with a small amount of read-modify-write for possible atomic operations. The predefined workloads A (update heavy) and F (read-modify-write) were selected out of six provided by YCSB framework, as they correctly imitate the behaviour of an average user program.

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

The benchmark consists of two phases: loading and transaction. In the loading phase, 1000 entries are inserted into each data-serving system. Figures 2.1a and 2.1b 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 might be due to the fact that Datastore, unlike the other to data-serving systems in question, is using synchronous replication, which makes the host wait until all replications are created, as described in Margaret's Rouse article Synchronous replication (Rouse, 2016).



(a) Latency (μs)



(b) Throughput (ops/sec)

Figure 2.1: Latency and throughput for 1000 insert (write) operations

In the transaction phase, two workloads were run. Workload A consists of 1000 operations (500 reads and 500 updates) while workload F consists of 2000 operations (1000 reads, 500 atomic read-modify-write operations and 500 updates). The results of the benchmark in terms of latency on write (update) operations were consistent with the previous loading benchmark results, with Bigtable and Spanner performing significantly better over Datastore (Figure 2.2). However, the gap between Bigtable and

Spanner was wider than the one observed in the loading phase, with Bigtable achieving the latencies of 4,627 μs on workload A and 4,682 μs on workload B, while Spanner struggling with the latencies of 9,944 μs and 7,338 μs, respectively.

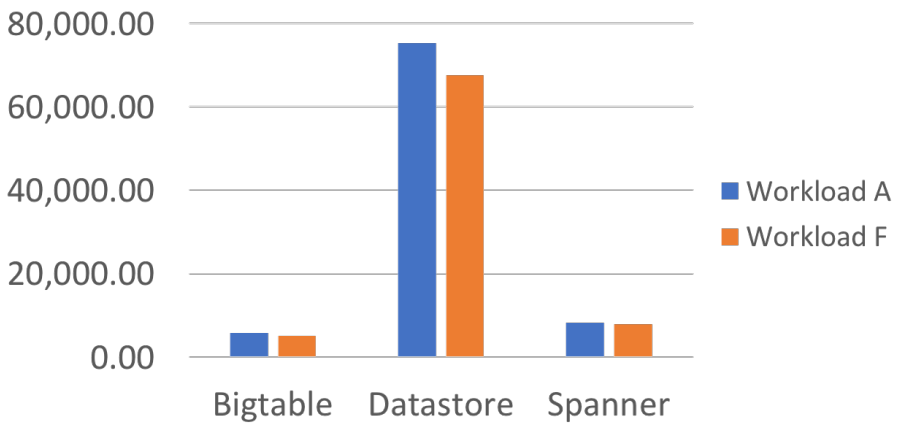


Figure 2.2: Update operations latencies (μs) for Workload A (500 writes) and Workload B (1000 writes)

Even though, the difference on read operations latency between Datastore and two other data storages were smaller than with write operations (Figure 2.3), 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 2.4).

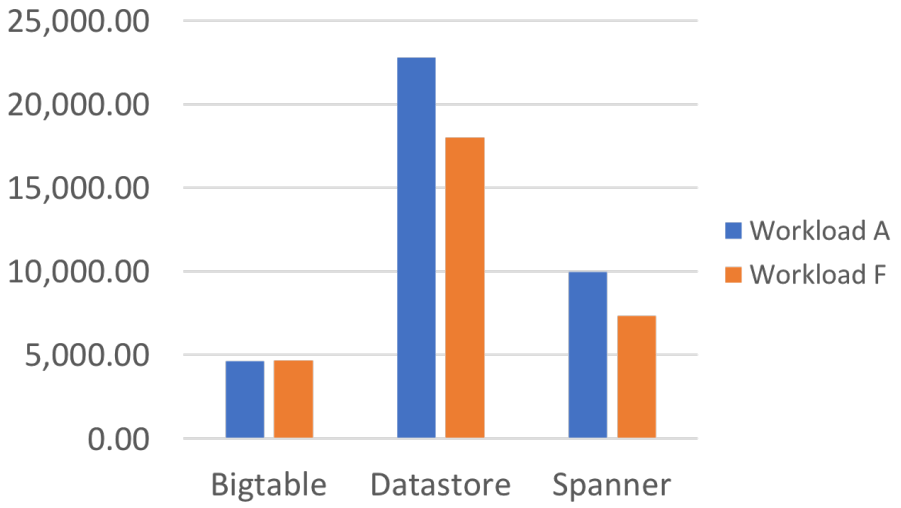


Figure 2.3: Read operations latency (μs) for Workload A (500 reads) and Workload B (1000 reads)

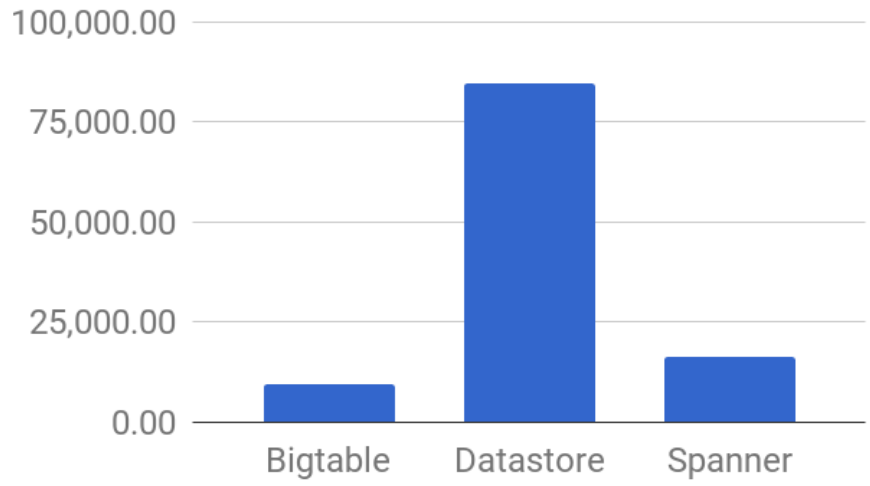


Figure 2.4: Read-modify-operations latency (μs)

The overall throughput for each workload, again, showed a significant superiority by Bigtable, as indicated in Figure 2.5. Bigtable achieved a throughput of 149.67 and 113.26 operations per second for workloads A and F, respectively. Spanner reached 105.63 and 78.19 operations per second, while Datastore performed worst with only 20.37 and 18.82 operations per second.

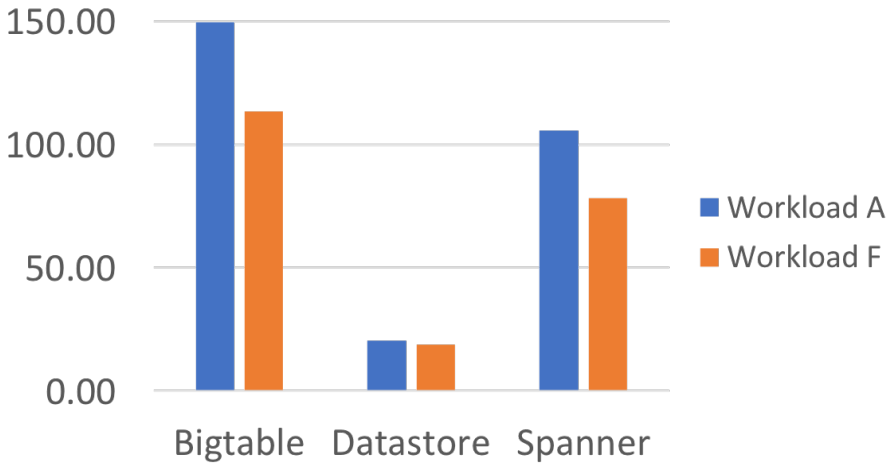


Figure 2.5: Throughput (ops/sec) for Workloads A (1000 operations) and Workload B (2000 operations)

The benchmark proved Bigtable to be fastest among the three selected data stores. The latency results for write and read operations are similar to the ones provided on Google Cloud Platform Blog post introducing Bigtable (O'Connor, 2015). Furthermore, the post claims Bigtable to provide cheaper write throughput (MB/s) per dollar performance than better read/write operations latency than their biggest competitors, namely HBase and Cassandra. Since, Bigtable meet all the requirements set for data

store and showed the best latency and throughput results, it was decided to use it as the main memory source for the programs instrumented with the translation tool. The next section gives a more in-depth look at Bigtable in order to better understand how it works and how it achieves such great performance.

2.3 Bigtable

Bigtable (Google, 2018a) is a high performance, wide column NoSQL database, designed to reliably scale to petabytes of data and thousands of machines. It is used by such Google products and projects like Google Analytics, Google Earth or Google Finance. Bigtable stores data in massively scalable tables, each of which is a persistent 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. Every read or write of data under a single row key is atomic. Moreover, Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned into tablets, which are the units of distribution and load balancing.

Bigtable uses the distributed Google File System (GFS) to store log and data files. It depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status. Bigtable data is stored using SSTable file format. Each SSTable contains a sequence of blocks. A block index is used to locate blocks; the index is loaded into memory when the SSTable is opened, allowing a lookup to be performed in a single disk seek. Optionally, an SSTable can be completely mapped into memory, which greatly increases the performance. However, such setup is very expensive and was not considered throughout the project.

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.

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

2.4 Reading and writing the contents of Bigtable

Bigtable uses gRPC client to read and write content using C++ remote function calls. According to gRPC webpage (Google, 2018b), gRPC client application can directly call methods on a server application on a different machine as if it was a local object (similarly to Java RMI). By default gRPC uses protocol buffers, Google's open source language and platform neutral mechanism for serialising structured data. The figure

2.6 visualises communication between Bigtable and its gRPC clients (stubs). Bigtable gRPC client is provided through Google APIs repository.

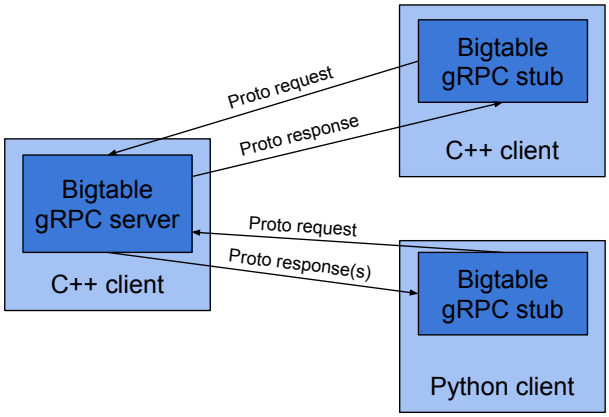


Figure 2.6: Communication between Bigtable gRPC server and Bigtable gRPC stubs (clients)

Get and put functions were implemented to abstract the specific details of the communication with Bigtable from the translation tool that calls them. This way the cloud storage service can be easily changed without the need to change the part that does the translation.

```
void put(unsigned long long addr, long long val)  {
  // cast arguments to string type
  string address = std::to_string(addr);
  string value = std::to_string(val);
  // setup the request
  MutateRowRequest req;
  req.set_table_name(tableName);
  req.set_row_key(address);
  auto setCell = req.add_mutations()->mutable_set_cell();
  setCell->set_family_name(familyName);
  setCell->set_column_qualifier(columnQualifier);
  setCell->set_value(value);
  // invoke row mutation on Bigtable
  MutateRowResponse resp;
  grpc::ClientContext clientContext;
  auto status = bigtableStub->MutateRow(&clientContext, req, &resp);
}
```

Listing 1: Writing content to Bigtable using put function

Put function takes two arguments, 64-bit integer as an address (key) and 64-bit integer as a value, and does not return anything (see Listing 1). First, the arguments are casted to string type. Then a new `MutateRowRequest` is built, by providing full path to the table (including project, instance and table names), row key, family name and column qualifier and value. Family name and column qualifier are constant as we are using Bigtable as key-value store, thus only one column family and qualifier is used. Finally,

the MutateRow function is called remotely through Bigtable stub and response status is stored for debugging purposes.

Get function takes 64-bit integer representing the address (key) as an argument and returns a 64-bit integer value (see Listing 3). Similarly to put function, the address value is cast to string. A read row request is created by providing the full path to the table and address string is passed as a row key. The call on ReadRows function returns a stream, which is read by chunks and appended to the valueStr variable. The nested loops should run at most one time, as only one row key was provided. Before the value is returned, an if statement checks if the given key had the corresponding value in the table and if so, casts the value to 64-bit integer. If no value was found with corresponding key, the function returns 0. This behaviour is expected, when the program reads uninitialised variable.

```
long long get(unsigned long long addr) {
  // convert argument to string type
  string address = to_string(addr);
  // setup the request
  ReadRowsRequest req;
  req.set_table_name(tableName);
  req.mutable_rows()->add_row_keys(address);
  string valueStr;
  // invoke row reading on Bigtable
  auto stream = bigtableStub->ReadRows(&clientContext, req);
    while (stream->Read(&resp)) {
      for (auto& cellChunk : *resp.mutable_chunks()) {
        if (cellChunk.value_size() > 0) {
          valueStr.reserve(cellChunk.value_size());
        }
        valueStr.append(cellChunk.value());
      }
    }
    // convert value to 64-bit integer
    long long value = 0;
    if (!valueStr.empty())
      value = stoll(valueStr);
    return value;
}
```

Listing 2: Reading content from Bigtable using get function

Additionally, atomic increment function was implemented, which given an address and a 64-bit integer increment invokes Bigtable's ReadModifyWriteRow method with the increment_amount rule. At first, the address is cast to string and set as a row key for ReadModifyWriteRowRequest. After request setup, read-modify-write operation is invoked. The respond message contains an array of bytes represented as an array of characters. A bytesToInt function was implemented in order to concatenate the bytes into a 64-bit value, which is then returned by the function.

```
long long atomic_increment(unsigned long long address, unsigned long long increment) {
  // convert address to string
  string address = to_string(address);
  // setup the request
  ReadModifyWriteRowRequest req;
  req.set_table_name(TABLE_NAME);
  req.set_row_key(address);
  ReadModifyWriteRule rule;
  rule.set_family_name(FAMILY_NAME);
  rule.set_column_qualifier(COLUMN_QUALIFIER);
  rule.set_increment_amount(increment);
  *req.add_rules() = std::move(rule);
  // invoke read-modify-write operation on Bigtable
  ReadModifyWriteRowResponse resp;
  grpc::ClientContext clientContext;
  auto status = bigtableStub->ReadModifyWriteRow(&clientContext, req, &resp);
  const char* bytes = resp.mutable_row()->mutable_families(0)->
                        mutable_columns(0)->mutable_cells(0)->value().c_str();
  long long value = bytesToInt(bytes);
  return value;
}
```

Listing 3: A read-modify-write operation with an increment rule on Bigtable using atomic_increment function

2.5 Issues encountered

Even though both gRPC and protobuf (protocol buffers) libraries are developed by Google, some difficulties were encountered while compiling source builds. The errors were made known to the developers (grp, 2017), but it has slightly stalled the development of the project.

Chapter 3

Translation tool

The translation tool was implemented as a compiler level solution mainly because it makes the tool portable across different platforms and it doesn't require any user code modification. One possible way to implement a compiler level solution is by building a new compiler, which instead of using load and store instructions would call get and put functions on Bigtable. Of course, creating a new, industry standard compiler would be too excessive and not feasible in the timespan of the project. Luckily, LLVM provides a compiler and lots of tools to instrument an already compiled code.

3.1 LLVM

LLVM (Low Level Virtual Machine), began at the University of Illinois, is an open source compiler framework for building tools. It supports life-long program analysis and transformation for arbitrary programs. It has an industrial standard compiler (clang/clang++), which has an option to compile C/C++ code to an extensible, strongly typed intermediate representation, namely LLVM IR.

LLVM optimizing compiler, like other industry standard compilers, consists of several components: frontend, optimiser, backend and linker. The important advantage over other compilers is the use of LLVM Bitcode, which consists of a bitstream container format and an encoding of LLVM IR. It provides a clean API boundary separating the compiler frontend and backend, thus making it easier to swap in new frontend and backend components (Figure 3.1). This is especially useful when developing a new language, as one only needs to create a new frontend component of the compiler and use the provided LLVM optimiser and backend components.

Moreover, as LLVM project is open-source, users can create their own optimisation or transformation passes. Thus it is possible to create an transformation pass for `opt` tool, which would iterate over all of the RISC-like instruction set and translate load and store instructions to get and put function calls for Bigtable.

LLVM has lots of support and documentation on the Internet. Lots of well-known compilers, like NVIDIA's CUDA Compiler or Microsoft DirectX shader compiler, are

based on LLVM. All of these features make LLVM a desirable framework to use for our translation tool.

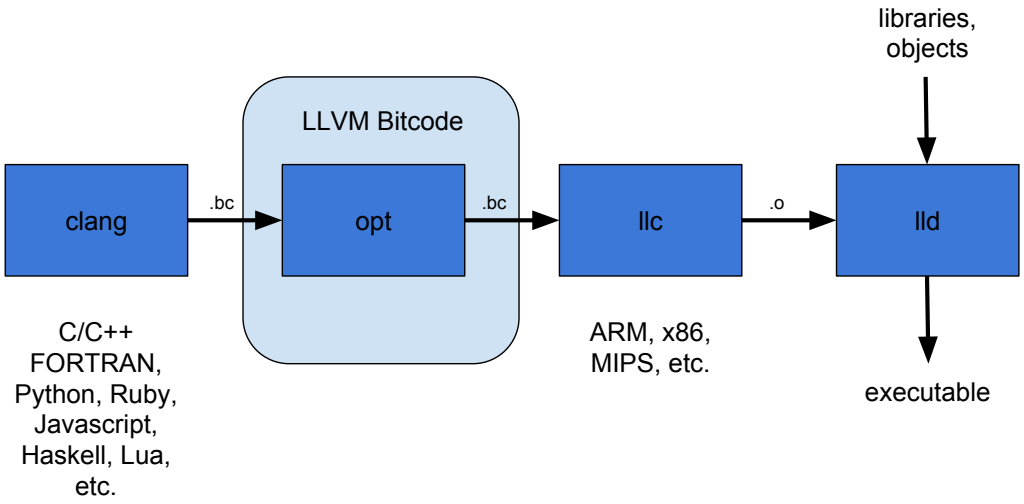


Figure 3.1: LLVM compiler architecture

One problem encountered when working with LLVM is that it cannot get LLVM IR representation of C library functions, as C library located on the machine is already compiled and assembled. This introduces a problem, as C library functions use load and store instructions to retrieve the data from the main memory, which after our translation is done will be stored in the cloud storage. The solution is to use portable C libraries like `ulibc` or `newlib`. They are C libraries available in source form intended for use on embedded systems. These libraries can be compiled into LLVM IR code, thus letting the translation pass transform load and store instructions.

3.2 Architecture

The translation pass tool consists of 4 transitions (see Figure 3.2). At first, user source code and Bigtable get and put functions are compiled to LLVM IR and combined into a single LLVM Bitcode file. As mentioned before, get and put functions of any other cloud storage could be used here. Additionally, any other C/C++ files which might be needed during the translation pass must be added at this stage. After that the code is instrumented by our transformation pass using `opt` tool and is emitted with translated LLVM IR into a LLVM Bitcode file. Then the output is compiled into native machine code using `llc` tool and finally linked with gRPC, protobuf and any other linked libraries to build an executable.

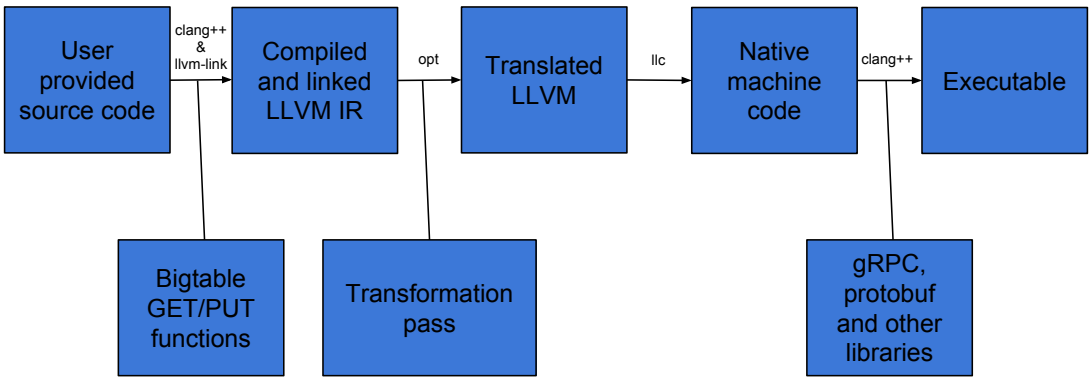


Figure 3.2: LLVM solution architecture

After the first transition when user source code and Bigtable functions are compiled and combined into a single LLVM Bitcode file, the resulting LLVM IR consists of dozens of gRPC library functions. These functions must not be translated, as they are responsible for setting up the connection and invoking the methods on Bigtable. Thus, a barrier was added to the transformation pass, which stops the transformation when the first gRPC function is detected through the iteration. In order for this to work properly, the linking of LLVM Bitcode files in first stage must be done in a strict order: Bigtable get and put functions file must appear after the code that must be translated[1]. This creates a barrier (see Figure 3.3) between the code that is translated and the code which contains functions to be called by the translated code (i.e. put and get functions).

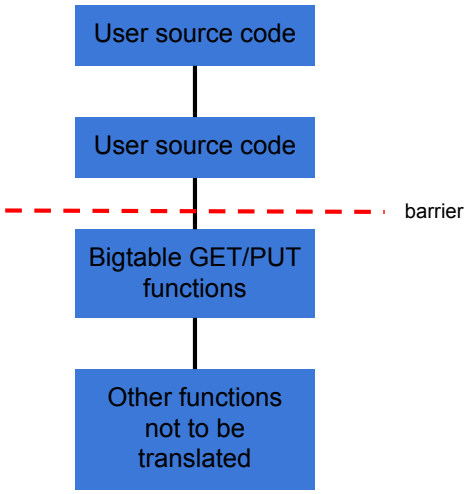


Figure 3.3: Translation barrier

[1]Notice that this implementation is only relevant for Google cloud storages. As other cloud services do not provide communication with storage using gRPC.

3.3 Translation pass

The previously mentioned barrier dividing instrumented and uninstrumented code is implemented by checking if the currently iterated function belongs to the original get and put functions file. The translation pass iterates over all instructions until the barrier. When the iteration reaches this function, the translation pass is ended.

Load and store instructions accept different types of arguments (remember, LLVM IR contains a strongly typed instruction set). For store instructions, the address must have a higher pointer indirection degree than the value. For instance, the corresponding store instruction for `int* x = &y` would have the address with the type of pointer to pointer (represented in LLVM as `**i32`) to integer and the value with the type of pointer to integer (`*i32`). For load instruction translations, get function must return the value of the same type as load instruction.

The LLVM load and store instructions accept any types of values for their operands. The same functionality for get and put functions could be achieved by creating multiple overloads. However, this would mean creating numerous overloads for each single type in LLVM, as a new overload function would be needed for each pointer type with different indirection degree. For instance, get function which takes pointer to 32-bit integer, would work with pointer to pointer to 32-bit integer. For this reason, get and load functions only accept 64-bit integer arguments and depend on additional instructions to enforce this type. This makes the translation of store and load instructions more complicated that just replacing one instructions with the other.

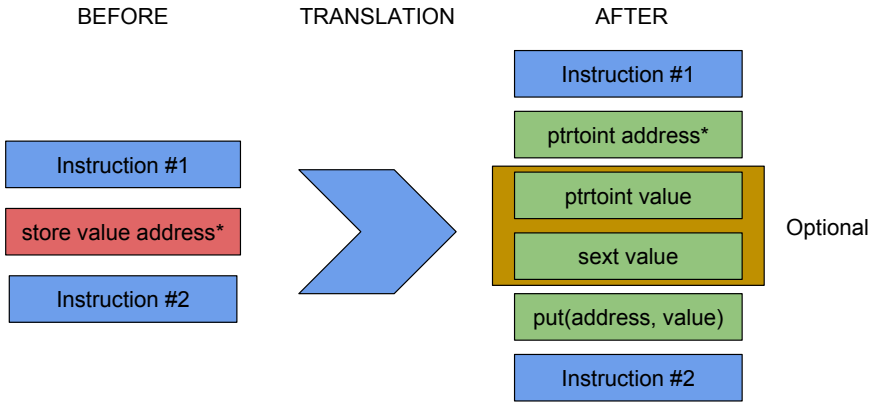


Figure 3.4: LLVM instruction set before and after store instruction translation

Store to put instruction translation starts by converting the address pointer to 64-bit integer with `ptrtoint` instruction. If the value is of integer type and not 64 bits wide, it is sign extended using `sext` instruction. If the value is of pointer type it is cast to 64-bit integer by `ptrtoinst` instruction. Otherwise, it is assumed to be 64-bit integer[2].

[2] Although it might be of some other type (i.e. struct). These types were not examined in the project and should be considered as future work.

Finally, both arguments now being of 64-bit[3] integer type are given as arguments to put function call. The figure 3.4 shows a subset of instruction set before and after store instruction translation.

Load instruction translation begins by identifying its return type and pointer indirection degree (only relevant if the value is of pointer type). Similarly to store translation, the address is cast to 64-bit integer using `ptrtoint` instruction, and passed to get function as an argument. As the return type of get function is a 64-bit integer, it must be cast to the appropriate type (unless the load instruction actually returns 64-bit integer). If the returned type is integer, it is truncated to the expected integer type using `trunc` instruction. Finally, if the expected returned type is pointer, the resulting value is converted to a pointer type with an appropriate pointer indirection degree (identified at the beginning) with `inttoptr` instruction. The figure 3.5 shows a subset of instruction set before and after store instruction translation.

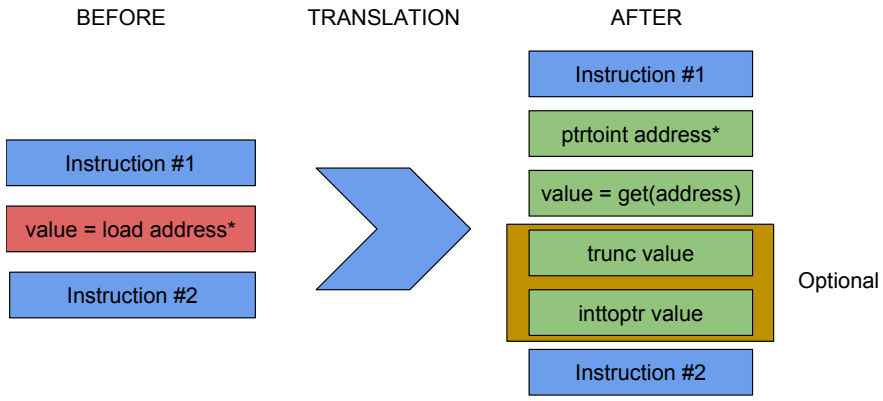


Figure 3.5: LLVM instruction set before and after get instruction translation

[3]64-bit integer type for get and put instruction arguments and/or return type was chosen deliberately. This is the maximum biggest type of integer, thus it makes the casting part a bit simpler. For example, if 32-bit integer type was chosen, some values would need to be sign extended while others would need to be truncated.

Chapter 4

Custom heap allocator

4.1 Motivation

Calls to malloc function allocate virtual memory space. When this allocated virtual memory is ”touched” (i.e. load from or stored to), it is mapped to real memory by call to special function of the operating system (i.e. mmap). As translation tool eliminates all loads and stores to memory, the allocated memory is never accessed, thus never really mapped to physical memory. Nevertheless, the allocated memory is left unused on virtual address space. This might cause problems when a program tries to allocate more space than possible on virtual memory. Some operating systems (i.e. Windows or macOS) limit the virtual memory space, thus providing a custom heap allocator, which would reduce the amount of memory allocated on virtual address space, seems beneficial. Moreover, a custom heap allocator can be implemented to provide information about the allocated heap boundaries for the program. The reason why this might be advantageous is explained in Chapter 5.

4.2 Implementation

4.2.1 First-fit free-list allocator

One of the proposed solutions is a first-fit free-list heap allocator. The heap allocator implementation was based on Marwan Burelle’s malloc tutorial (Burelle, 2009) and adjusted to work on Bigtable. The heap allocator implements four functions: malloc, free, realloc and calloc. All of these functions are counterparts of the Standard C++ Library functions and have the same function definitions.

The allocator keeps a list of meta-data blocks, which contain information about chunks of data allocated with the new malloc. This lets the memory be reused after it has been released with a call to free function. Unlike in tutorial implementation, only meta-data objects are stored on main memory. The actual requested space is allocated on Bigtable. Thus, the implemented custom heap allocator uses two heaps: main memory

and Bigtable. The figure 4.1 sketches the memory organisation of two heaps. The structure of meta-data blocks described in the tutorial were adjusted to reflect these changes. Besides storing the size of data block and pointers to other blocks, the meta-data blocks also store the address of data block on Bigtable.

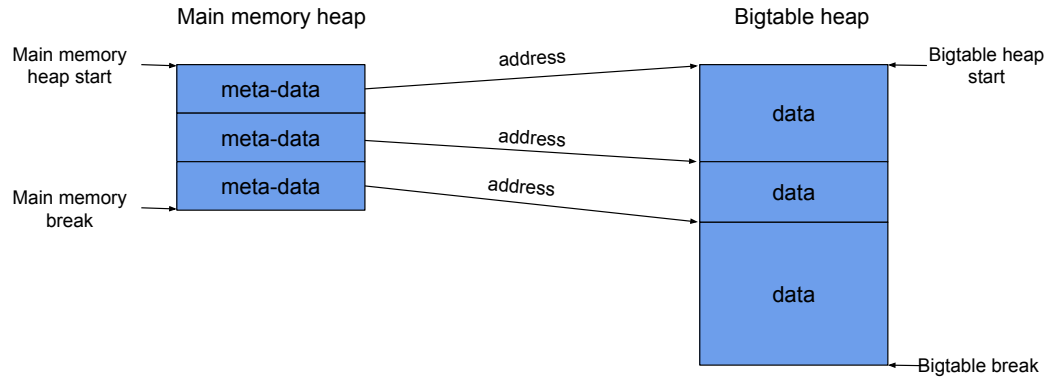


Figure 4.1: Heap memory organisation using the proposed heap allocator

The main memory heap is managed by the default memory allocator (Standard C++ library provided malloc, free, etc). All of the malloc (and other memory allocation functions) calls in the user source code are translated to custom heap allocator function calls using the translation pass described in the previous chapter. Other memory allocation function calls (i.e. in gRPC functions) are not translated and use the default allocation functions.

The heap on Bigtable is implemented as a continuous space of memory with two bounds: the start of the heap and the end point called the Bigtable break. The start of the heap is initialized on the first call on custom malloc function by calling `sbrk` function with an increment equal to 0 (this returns the break address on the main memory heap). Thus, the start addresses of main memory and Bigtable heap are identical. The end of the heap is managed by set_bt_brk function, which was implemented to reflect the behaviour of `sbrk` function (see Listing 4).

```
void* set_bt_brk(int incr) {
  // if uninitialised, set to sbrk(0)
  if (current_bt_break == 0) {
    current_bt_break = (uintptr_t) sbrk(0);
    initial_bt_break = current_bt_break;
  }
  uintptr_t old_break = current_bt_break;
  current_bt_break += incr;
  return (void*) old_break;
}
```

Listing 4: set_bt_brk function implementation

An alternative way to implement such heap allocator is to allocate both meta-data objects and the actual requested space on the Bigtable, but this would increase the

communication costs with Bigtable. As meta-data objects only take up to 40 bytes (on a 64-bit system), the decision was made to store them locally.

Malloc function starts by changing the requested size to be a multiple of 4 to align the pointers by 32 bits. If the list of meta-data blocks is not empty (meaning custom malloc, calloc or realloc was called before), the linked list of meta-data blocks is searched for the first free chunk that is wide enough for the request. If such block is found and the difference between the requested size and the size of the block is enough to store a minimum allowed block (32 bytes[1]) the block is split into two blocks. The first block is marked as used, while the second one is added to the linked list. If the list of meta-data blocks is empty or no existing wide enough block is found, Bigtable heap is extended by calling set_bt_brk and creating a corresponding meta-data block for the allocated space. Finally, the address to the block on Bigtable is returned.

Free function accepts a pointer to heap memory block to be freed and starts by checking if the address it points to is a valid Bigtable heap address. If it is, the address is used to retrieve and mark the corresponding meta-data block as free. If the any of the neighbouring meta-data blocks are free, the blocks are fused into one to cope with fragmentation. If after the fusion the resulting meta-data block is the tail of the linked-list, both the memory on Bigtable and the meta-data block are released. This means that the Bigtable break is pushed back by the size of the block being released and the meta-data block is freed using a default free function.

As mentioned above, the implemented free function needs to retrieve the meta-data block using the given Bigtable address. One way to do this is by iterating the linked list of meta-data blocks and checking if its address to block on Bigtable matches the given address. This is a rather inefficient solution ($\Theta(n)$, where n is the number of meta-data blocks). In order to do this efficiently a hash table was introduced to map the address of block on Bigtable address to meta-data block address. Every time a heap is extended or a block is split, the hash table is updated with a new key-value (Bigtable block address, meta-data block address) pair. The hash table improves time efficiency ($\Theta(1)$) in exchange for some space ($\Theta(n)$).

Realloc function accepts two arguments, pointer to an existing Bigtable heap memory block and the new size for the block. If the pointer address is a valid heap address, the meta-data block is fetched using the hash table mentioned above. The size is changed to be aligned with 32-bit pointers. If the requested size is smaller than the original size, the block is split in two. Otherwise, if the next block is free and provide enough space (combined with the original block), the two blocks are fused into one and split if contains more space than required. If none of the options above are true, a new block is allocated with the custom malloc, the old block contents are copied to the new one and, finally, the old block is freed. The block copying procedure was modified to work with Bigtable. The old block values are copied using a combination of get and put function calls to Bigtable (see Listing 5). Lastly, if the pointer given as an argument to realloc is null, the behaviour is the same as calling custom malloc.

[1] In the popular dlmalloc (Lea and Gloger, 1996) implementation, the smallest allowed allocation is 32 bytes on a 64-bit system, thus it was decided to use the same size in the proposed malloc implementation.

```
void copy_block(block src, block dst) {
  int *sdata, *ddata;
  unsigned long long value, *a, b;
  sdata = (int*) src->addr;
  ddata = (int*) dst->addr;
  for (size_t i = 0; i*4 < src->size && i*4 < dst->size; i++) {
    // convert int* to 64-bit integer
    a = (unsigned long long*) &sdata[i];
    b = (unsigned long long) a;
    value = get(b);
    // convert int* to 64-bit integer
    a = (unsigned long long*) &ddata[i];
    b = (unsigned long long) a;
    put(b, value);
  }
}
```

Listing 5: copy_block implementation

Calloc function accepts an integer representing a number of elements to allocate and an integer representing the size of each element. First, the custom malloc is called with the product of two arguments. The new block is iterated by 32-bit steps and initialised with 0 values. Again, the implementation was modified to work with Bigtable, using put function (see Listing 6).

```
  s4 = align4(num * size) >> 2;
  for (i = 0; i < s4; i++) {
    // convert int* to 64-bit integer
    unsigned long long* a = (unsigned long long*) &new_block[i];
    unsigned long long b = (unsigned long long) a;
    put(b, 0ULL);
  }
```

Listing 6: new_block initialisation with zeroes

4.2.2 Allocator with no memory reusage

Another proposed solution is a heap allocator with only a malloc function, without any memory releasing strategy. Heap memory address is incremented with an atomic read-modify-write operation. Calls to free function effectively do nothing, while calls to calloc or realloc imitates the behaviour of malloc. Such memory allocator cannot reuse the previously freed memory, which results in memory leaking. However, in theory, it should work faster than the default and first-fit free-list allocators.

To use the implemented allocators instead of the default one, additional translation operations were added. They change the default malloc call instructions to the implemented malloc call instructions. The heap allocator source code is provided to the tool during the first transition, mentioned in Section 3.2.

4.3 Consistency on multithreaded programs

Even though the malloc tutorial helped a lot to implement a first-fit free-list heap allocator, it didn't mention anything about allocator consistency on multithreaded programs. The simplest solution to this problem was implemented using a lock. A single mutex was created and the calls to its functions, `lock` and `unlock`, were added to the entry and exit points of malloc, free, realloc and calloc functions. The added synchronisation prohibits other threads from doing allocations and releases while the other thread is modifying the heap. Even though this is a correct solution, it is very inefficient. Threads that frequently allocate and release memory from heap are constantly being blocked or block others threads, essentially making the execution of the program serial.

Another way to solve the thread-safety problem is by allocating a large chunk of memory off the heap to each thread and then managing the space within the thread. However, some threads might not be using all of their allocated memory, which results in poor memory usage efficiency. Moreover, some threads might need more memory than given by default, thus there should be ways to increase the per thread heap memory. It can be seen that the increase in performance adds additional complexity to the allocator. Clearly, this is a more involved solution and due to a strict time limits for the project it was not implemented. This is one of the areas where the translation tool could be improved in the future work.

On the other hand, the allocator with no memory reusage is thread-safe and requires no additional synchronisation. This is achieved by an atomic add operation provided by C++ standard library.

4.4 Results

4.4.1 Virtual memory usage

Both heap allocators implemented were tested for virtual memory usage. The figure 4.2 shows Linux `top` command output for memory usage on an instrumented program without heap allocations and with three different heap allocators. The top left screenshot shows a program run without any heap allocations. It is used as a baseline run to better understand how much memory an instrumented program uses. Each of the other 3 processes malloced 1 gigabyte of memory. Clearly, the processes using the proposed heap allocators solve the problem of virtual memory usage, as the allocated gigabyte of memory does not show up on the process memory usage statistics.

DATA	VIRT	RES	SHR	COMMAND
84448	693308	17744	10072	snapd
27812	268804	13052	10152	run
26604	260768	3744	3040	rsyslogd

(a) run without malloc

DATA	VIRT	RES	SHR	COMMAND
1076392	1317384	12948	10044	run
84448	693308	17744	10072	snapd
26604	260768	3744	3040	rsyslogd

(b) run with default malloc

DATA	VIRT	RES	SHR	COMMAND
84448	693308	17744	10072	snapd
27812	268804	12912	10020	run
26604	260768	3744	3040	rsyslogd

(c) run with custom malloc

DATA	VIRT	RES	SHR	COMMAND
84448	693308	17736	10072	snapd
27812	268796	12536	9724	run
26604	260768	3744	3040	rsyslogd

(d) run with custom malloc without free

Figure 4.2: The Linux top command output for memory usage on 4 different instrumented programs

4.4.2 Performance

Several different performance tests were carried out in order to measure allocators efficiency on different scenarios. All tests were rerun 100 times with the produced times averaged. Test A allocates 50 chunks of 1kB data and frees it, while Test B allocates 50 different sized (from 1kB to 50kB) chunks of data. Figure 4.3 shows the times taken to finish tests A and B with the three allocators. The default allocator seems to work very efficiently with single sized data but performed worst on Test B with different sized data. First-fit free-list allocator similarly with both tests, achieving an average of 46.5 μs. Allocator without memory reusage achieved the best times on both tests, averaging 1 μs.

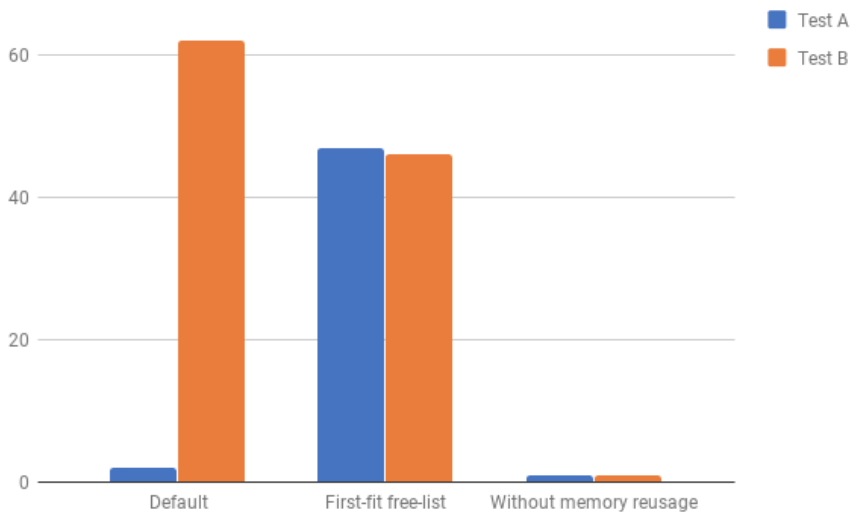


Figure 4.3: Test A and B clock time (μs) results for three allocators

Test C allocates a few blocks of 512 bytes of memory, frees them and repeats this loop 10 times. Test D allocates 50 block of 1kB data, releases half of it and then allocates and frees memory in a loop. Each test was run twice: once with single size blocks and once with different size blocks. Figure 4.4 shows the times taken to finish

tests C and D with different allocators. Once more the results show that the default allocator is struggling with different size blocks, while performing very good with single size blocks. First-fit free-list allocator seems to perform badly on Test D with different size blocks. The poor performance can be explained by one little detail in test D: half of the primarily allocated memory is released by freeing every other block. This means that the first-fit free-list allocator cannot fuse the freed blocks and suffers from fragmentation. On the other hand, the allocator without memory reusage is again performing outstandingly, averaging 11.5 μs for each test.

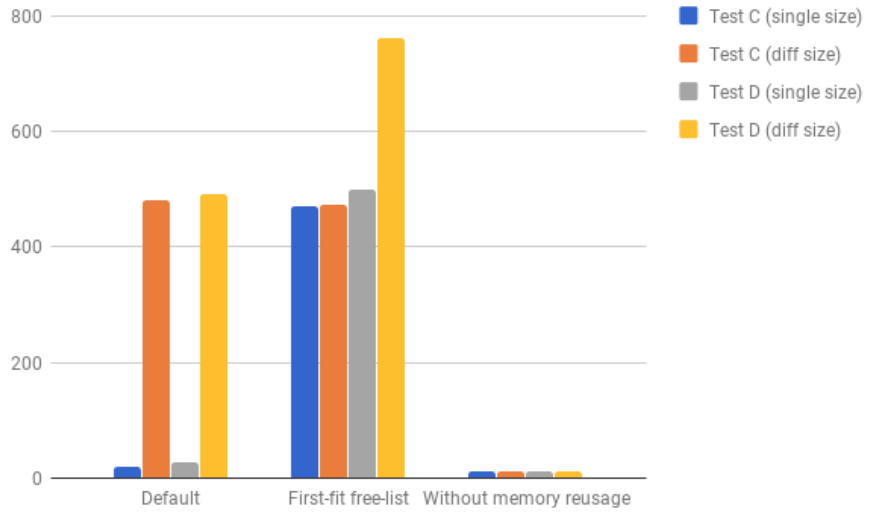


Figure 4.4: Test C and D clock time (μs) results for three allocators. Single size means that the test was run with single size blocks, while diff size means that the test was run with different size blocks.

Finally, the last two tests perform tests A and B in parallel. First test A is run with 2 threads, then with 10 threads. The same is done with test B. Figure 4.5 shows the performance achieved by different allocators during these tests. The default allocator shows quite good times on tests A and B with 2 threads and test A with 10 threads but performs poorly on test B with 10 threads, which confirms the poor performance on different size memory blocks allocation. As explained in Section 4.3 and proven in the chart, synchronisation using locks severely punishes the performance of first-fit free-list allocator, thus it is no surprise it performs worst. The allocator without memory reusage, however, repeatedly performs best.

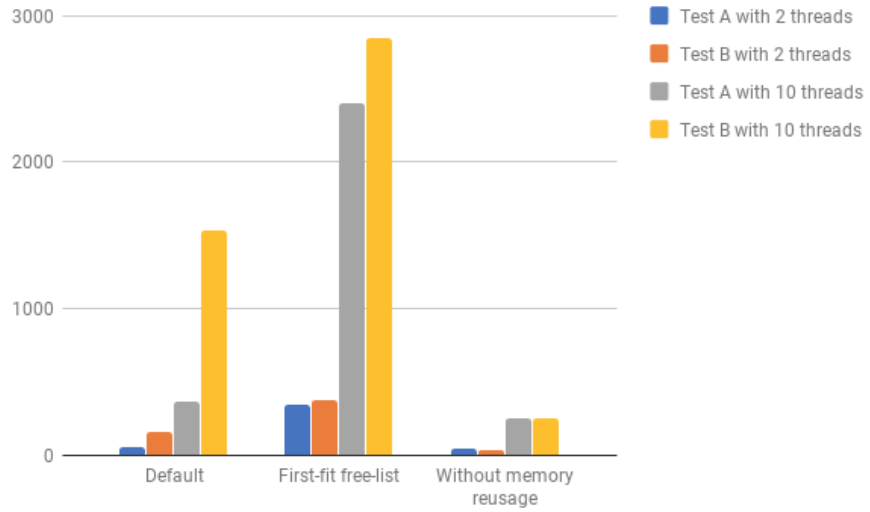


Figure 4.5: Test A and B performance results when run in with 2 and 10 threads.

Overall, the results show that the default allocator works faster than the first-fit free-list allocator but the difference in single-threaded different size block allocations is not tremendous. An allocator without memory reusage seems to work best and might be a better solution for short running programs.

Chapter 5

Heap memory translator

5.1 Motivation

The translation tool introduced in Chapter 3 translates all store and load instructions. However, the translated store and load instructions which operate on stack, BSS or data segments do not actually help save main memory space as the memory for the variables on these segments is still being allocated even though the values are stored on and loaded from cloud storage. This means that only heap memory ”translation” gives the desired results and other program memory segments need more complex changes. For instance, BSS and data segments, which consist of initialised and uninitialised static variables, could be stored in cloud storage without allocating space on main memory by implementing a custom compiler frontend. Unfortunately, creating a custom compiler frontend did not seem feasible given the timespan of the project. Instead, a new version of translation tool, which translates only load and store instructions operating on heap memory, was created. The programs translated with the heap memory translator run faster, as it translates only a fraction of the existing store and load instructions. For instance, the tool does not translate such costly variables as loop iterators, which generate vast amounts of load and store instructions in programs.

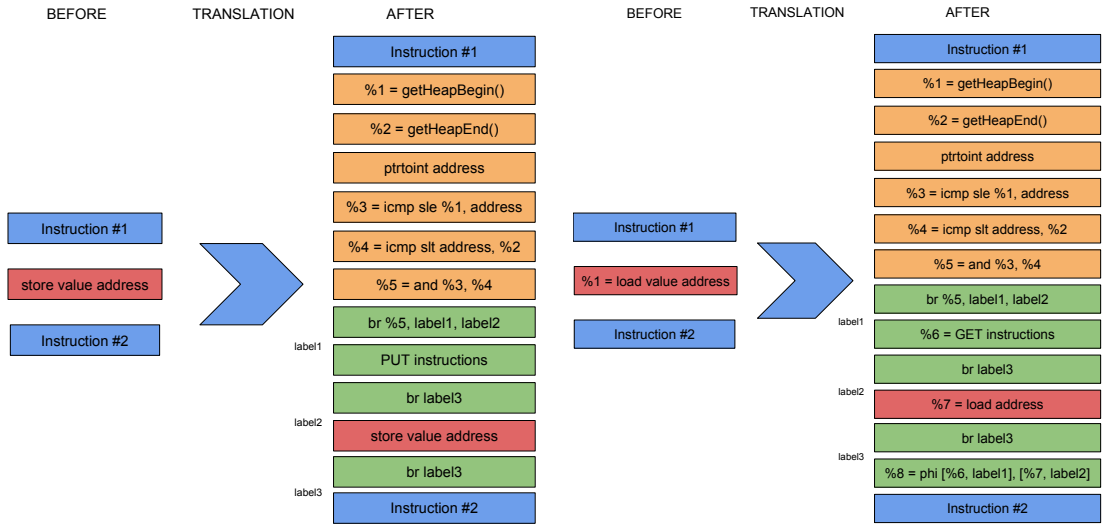
5.2 Implementation

The easiest way to determine if the variable is stored on the heap is to check if its address is inside the heap address space. If the address of the variable is between the start of the heap and the current break of the heap, then we want to store such variable on the Bigtable. Thus, while iterating through instructions we need to add checks to detect if the address of load or store instruction is inside the heap address space. If it is, we call get or put function on Bigtable and do the similar type casts as in full memory translator. Otherwise, we want to leave the original load or store instruction as is.

Firstly, the first-fit free-list heap allocator was extended to have two new functions. The getter functions for heap boundaries. In order to accomplish this, the allocator

saves the start of the heap on the first call to malloc.

For both store and load instructions the translation starts with calls to get heap boundaries. The address of the instruction, which is being translated, is cast from pointer type to 64-bit integer and compared using `icmp` instructions with both heap boundaries. The comparison results are conjoined bitwise and used as a condition for branch instruction. In a case where the bitwise conjunction is equal to 1, the program continues with the instructions for get or put function. Otherwise, the program jumps to the label with the original load or store instructions. If the translated function is load, then a PHI node is added to take on the value corresponding to the input control block (either call to get function or load instruction). PHI node is an instruction used to select a value depending on the predecessor of the current block.



(a) Store instruction translation (b) Load instruction translation

Figure 5.1: Load and store instruction translation on heap memory translator. Instructions coloured in orange and green are added during the translation.

As seen in figure 5.1, heap memory translation adds a lot of additional instructions. The insertion of instructions is done in several phases. On the first phase, the first six instructions (coloured in orange) are inserted before the original load or store instruction. Next, the original basic block, the load or store instruction belongs to, is split in two. Between them two additional basic blocks are created. All this is done using a `SplitBlockAndInsertIfThenElse` function. After that, the first basic block (label1) is populated by adding a call to get or put function and the casting instructions if needed. The second basic block (label2) is populated with a clone of the original load or store instruction, as the original still exists on the next basic block (label3). Then, the original store instructions are deleted from the last basic block (label3), while the original load instructions are replaced with PHI node instructions. Finally, heap allocator function translations are performed.

Chapter 6

Evaluation

Even though the translation tool introduced in this paper does not aim to compete on performance with existing technologies, it is still interesting to compare the original programs with the ones instrumented with the translation tool. Of course, the efficiency of the system which stores and loads all its memory on another machine, however fast the connection between two machines, is not going to compete with the system storing and loading memory situated on the same machine. Nonetheless, the performance and other results can be of great value, as they reveal the areas where the tool could be improved.

During the testing and evaluation process, it was decided to measure the performance of individual functions rather than the execution of the whole program itself. This decision was taken in order to exclude the time it takes to load the program into memory and prepare it for execution. These measurements are not affected by the translation tool and are highly depended on the architecture of the machine and the operating system. For the best performance results, the instrumentation tool was tested on Google Compute Engine virtual machine situated on the same data centre as Bigtable instance.

6.1 Performance

For the performance benchmarking, wall clock time was used instead of CPU time, as it takes into account the time taken for I/O operations (i.e. calls to Bigtable). All tests are run 100 times and averaged, as the execution time of a single test run fluctuates due to external factors (i.e. OS scheduling, memory usage, etc).

The benchmarking was done using three test functions (see Appendix A), namely stack allocation, heap allocation and parallel producer-consumer problem adapted from Greg Andrews' textbook website (Andrews, 2000). The stack allocation test initialises a stack-based 10x10 character matrix with new values. All load and store instructions operate on stack memory. Similarly, the heap allocation test writes values to heap-allocated 10x10 character matrix. Approximately one of ten load and store instructions are for heap-based accesses. Other, stack-based, memory operations are used for iterat-

ing through the matrix. The producer-consumer test uses C's POSIX threads (Pthreads) library for thread creation and synchronisation. For this task, the time elapsed includes thread creation and join operations. Two threads were used: one for producer and one for consumer operations. The buffer size was set to 1, while, in total, 50 integers were produced/consumed by the algorithm. The buffer is allocated on the heap in order to have at least one get or put function call when translated with heap translator.

As expected the original program with load and store instructions operating on the main memory of the machine worked efficiently. The producer-consumer took more than a thousand times longer than the other two tests, as the execution time was capped by thread creation and synchronisation time. The stack allocation and heap allocation tests translated with full memory translator took six orders of magnitude longer to execute. This immense increase in time is due to the introduced communication costs to access memory on Bigtable. The producer-consumer problem took half as long to execute, yet the increment from the original code execution time was enormous. Unsurprisingly, the program translated with heap translator showed a huge improvement over the fully translated program in stack allocation test. The improvement is quite clear as none of the load or store instructions are translated, as only heap memory instructions are translated with heap translator. The slow down compared to the original program can be explained by the increase in instruction count of the program (more detailed analysis on this measure provided in the next section). The heap allocation test on the program transformed by heap translator performs 10 times better than the corresponding test instrumented by full memory translator. The test is capped by the time taken to execute 100 put operations with heap memory values on Bigtable. Producer-consumer test again saves some time on load and store instructions on stack variables. The results gathered from benchmarking are shown in table 6.1.

Test	Original	Fully translated	Heap translated
Stack allocation	0.411	2,798,858	3
Heap allocation	0.521	2,209,032	284,047
Producer-Consumer	963.717	1,272,932	295,448

Table 6.1: Time taken to execute each function in microseconds (μs)

Overall, it can be seen that the heap translation tool is an improvement over full memory translator but it is still makes the functions run hundreds (producer-consumer test) and even hundreds of thousands (heap allocation test) times slower than their original counterparts.

6.2 Instruction count

To count the instructions of a program an instruction count pass was implemented. The pass iterates over the functions of the program and prints the count of instructions per function and the total number of instructions in the program. To clarify, this is a static count of the instructions and does not account for how many times any particular instruction can be executed on the runtime. Moreover, the instruction count does not

include the instructions from shared library functions. The instruction count pass was run on the test program (see Appendix A).

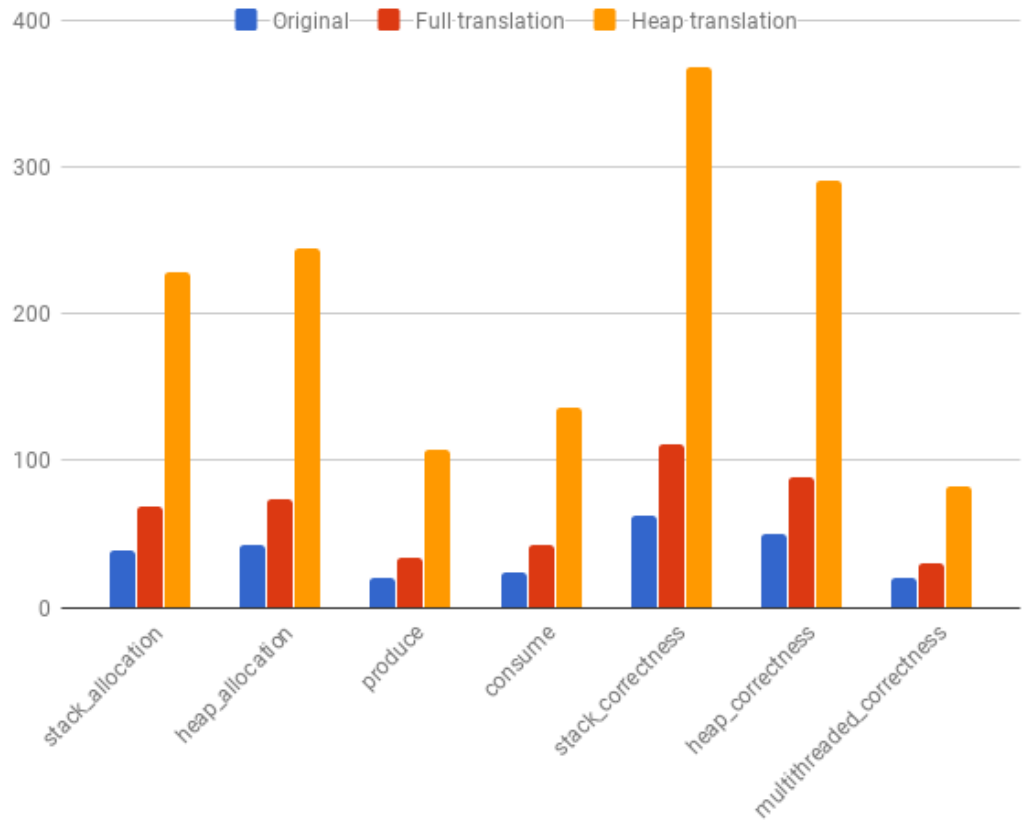


Figure 6.1: LLVM instruction counts for different functions.

The original program shows the smallest instruction count on every function and works as a baseline for the programs instrumented with the implemented translation tools. The total instruction count for a program translated with full memory translator increased from 571, achieved by the original program, to 761, which shows a 33 percent growth. On average, each function increased by 27 instructions. The inflation is a result of translating one load/store instruction with a call to get/put function instruction plus the additional 1-3 casting instructions. A much more substantial growth can be seen in functions translated by heap memory translator. The total amount of instructions is 209 percent higher than in the original program, increasing from 571 to 1768 instructions. Each function, on average, increased by 171 instructions. This can be explained by the additional control flow added to distinguish between memory access to heap and stack variables. The translation adds 9 to 14 instructions per translated store/load instruction.

Overall, the translation adds a lot of additional instructions. This does not necessarily convert to lower performance of the program (as can be seen by heap translation example with stack allocation test in the previous chapter) and is a much smaller problem

then the huge increases in execution time, yet it is definitely a measure which could be improved by additional heuristics.

Chapter 7

Conclusions

7.1 Overview

This report has introduced a translation tool (with two of its variations), which provides a way to run programs requiring amount of memory that cannot be served by user machine. The translation tool instruments the program by changing all load and store instructions to get and put calls on Bigtable, respectively. Additionally, another variation of the translation tool was implemented, which translates only those store and load instructions which operate on heap memory. The programs instrumented by heap translation tool run faster than the ones instrumented with full memory translator, as it translates only a fraction of the existing store and load instructions. On other hand, heap translator significantly increases the amount of instructions used in the program. Overall, the report showed that the performance of the instrumented programs are orders of magnitude worse than the original programs, thus there are still lots of things that could be improved.

7.2 Future work

One major area of future work might be to implement a separate front-end compiler. The compiler could make use of the implemented get and put functions when initialising global (static) variables, which in other case would be allocated space on main memory. Somewhat similar (although more complex) strategy could be implemented to prevent stack variables from being allocated space on main memory.

A major performance improvement could be achieved by having the program use main memory and start using Bigtable only when main memory is not enough to serve the requests. One way to achieve this could be incorporating dynamic translation, which would determine when to start use Bigtable based on some system interrupts indicating lack of memory problem.

There might also be a way to slightly improve the performance of the translated pro-

grams by introducing caches. They would store frequently accessed data on main memory, thus amortising the communication to Bigtable costs on most frequent values.

Other possible improvements or extensions on the tool:

- Better support for getting and putting structured data to Bigtable. For instance, C++ standard library class Tuple is represented as a literal struct type ({`i32, *i32`}). Load and store instructions provided by LLVM cope with these type of structures, yet the control flow created by the translation tries to cast the type into 64-bit integer and loses parts of data (32 most significant bits) on the cast back.
- A get and put function overloads should be provided for ConstantFP (floating point) type in order to correctly store floating point values on Bigtable.
- Introduce synchronisation through Bigtable between multiple instrumented programs running on different machines. The synchronisation could be achieved by using locks, barriers and other primitives. Bigtable provides an atomic fetch-and-add operation, which can be used as a generalised test-and-set operation, thus making it possible to implement the synchronisation primitives stored there. Combined with cache implementation and an appropriate cache coherence protocol the updates would let expose Bigtable as distributed shared memory, thus introducing a new usage of the tool.

Combined with all these improvements, the translation tool implemented on this project could be a way to solve lack of memory problem or even expose cloud storage as distributed shared memory.

Bibliography

Github. gRPC reporsitory. Issues. grpc c++ fails to compile (zlib), December 2017. URL https://github.com/grpc/grpc/issues/13640.

Eric A. Anderson and Jeanna M. Neefe. An exploration of network ram. Technical report, Berkeley, CA, USA, 1998.

Greg Andrews. Foundations of multithreaded, parallel, and distributed programming. Source code for programs, January 2000. URL http://www2.cs.arizona.edu/people/greg/mpdbook/programs/.

Marwan Burelle. *A Malloc Tutorial*. Laboratoire Systeme et Securite de l'EPITA (LSE), February 2009.

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*, pages 143–154, 2010. URL http://bit.ly/benchmarking-cloud-serving-systems-with-ycsb.

Jeff Dean. Designs, lessons and advice from building large distributed systems, 2009. URL http://bit.ly/numbers-everyone-should-know.

goldilocks. Can linux "run out of RAM"?, September 2016. URL https://unix.stackexchange.com/a/92544.

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

Google. gRPC, 2018b. URL https://grpc.io/docs/guides.

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.

Doug Lea and Wolfram Gloger. A memory allocator, 1996.

Cory O'Connor. Announcing google cloud bigtable: The same database that powers google search, gmail and analytics is now available on google cloud platform. *Google Cloud Platform Blog*, May 2015. URL http://bit.ly/introducing-google-cloud-bigtable.

J. Oleszkiewicz, Li Xiao, and Yunhao Liu. Parallel network ram: effectively utilizing global cluster memory for large data-intensive parallel programs. In *International*

Conference on Parallel Processing, 2004. ICPP 2004., pages 353–360 vol.1, Aug 2004. doi: 10.1109/ICPP.2004.1327942.

Margaret Rouse. Synchronous replication, 2016. URL `http://bit.ly/synchronous-replication`.

Appendix A

Test program

```
1  const int M_X = 10;
2  const int M_Y = 10;
3  const int N = 50;
4  volatile int target;
5  int* data;
6  char *heap_matrix;
7
8  void test_stack_allocation(char matrix[][M_Y]) {
9    for (int i = 0; i < M_X; i++) {
10     for (int j = 0; j < M_Y; j++) {
11       matrix[i][j] = (i * M_Y + j) % 128;
12     }
13   }
14 }
15
16 char *test_heap_allocation() {
17   for (int i = 0; i < M_X; i++) {
18     for (int j = 0; j < M_Y; j++) {
19       *(heap_matrix + i * M_Y + j) = (i * M_Y + j) % 128;
20     }
21   }
22 }
23
24 void *produce(void *arg) {
25   for (int i = 1; i <= N; i++) {
26     sem_wait(&empty);
27     *data = i;
28     sem_post(&full);
29   }
30   return NULL;
31 }
32
```

```
33 void *consume(void *arg) {
34   int total = 0;
35   for (int i = 0; i < N; i++) {
36     sem_wait(&full);
37     total += *data;
38     sem_post(&empty);
39   }
40   return NULL;
41 }
42
43 bool test_stack_correctness(char matrix[][M_Y]) {
44   for (int i = 0; i < M_X; i++) {
45     for (int j = 0; j < M_Y; j++) {
46       if (matrix[i][j] != (i * M_Y + j) % 128) {
47         cout << i * M_Y + j << "  " << matrix[i][j] << endl;
48         return false;
49       }
50     }
51   }
52   return true;
53 }
54
55 bool test_heap_correctness(char *matrix) {
56   bool correct = true;
57   for (int i = 0; i < M_X; i++) {
58     for (int j = 0; j < M_Y; j++) {
59       if (*(matrix + i * M_Y + j) != (i * M_Y + j) % 128) {
60         correct = false;
61       }
62     }
63   }
64   free(matrix);
65   return correct;
66 }
67
68 void *test_multithreaded_correctness(void *arg) {
69   for (int i = 0; i < N; i++) {
70     sem_wait(&lock);
71     target++;
72     sem_post(&lock);
73   }
74   return NULL;
75 }
```