COMPUTER ON-DEMAND ARCHITECTURE (CODARC)

An Undergraduate Research Scholars Thesis

by

HUNG QUOC BUI and DANIEL ERIC SCHWARTZ

Submitted to the Undergraduate Research Scholars Program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Sunil P. Khatri

May 2020

Majors: Computer Engineering
Electrical Engineering

TABLE OF CONTENTS

ABSTRACT

Computer On-Demand Architecture (CODArc)

Hung Quoc Bui and Daniel Eric Schwartz
Department of Electrical and Computer Engineering
Texas A&M University

Research Advisor: Dr. Sunil P. Khatri
Department of Electrical and Computer Engineering
Texas A&M University

This paper proposes a new computer architecture that is designed to be specifically lean and secure. Due to recent advances in network speeds which allow users to pull data from the internet just as fast as from local hard drive, this architecture forgoes the typical arrangement of computers and sources its hard disk data from the cloud. On the local hardware, the MMU is removed and local peripherals are implemented on an FPGA to be swapped out when not in use. We have completed most of the high-level design of the system, and have prototyped a suitable data server; however, the local software and hardware will need to be further developed for this architecture to be complete.

ACKNOWLEDGMENTS

We would like to thank our research advisor, Dr. Sunil Khatri for his guidance and support throughout the course of this research.

NOMENCLATURE

ACID	Atomicity, Consistency, Isolation, Durability
BIOS	Basic Input and Output System
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
HDMI	High-Definition Multimedia Interface
LUT	Lookup Table
MMU	Memory Management Unit
NFS	Network File System
OID	Object Identifier
OS	Operating System
PR	Partially Reconfiguration
PRC	Partial Reconfiguration Controller
RAM	Random Access Memory
TCP/IP	Transmission Control Protocol over Internet Protocol
TFTP	Trivial File Transfer Protocol
USB	Univeral Serial Bus

CHAPTER I

INTRODUCTION

In recent years, network speeds have increased rapidly allowing for innovations in internet-based functionality known collectively as the cloud. This cloud has been a major topic of research for over a decade now, typically focusing on storage systems. However, researchers are constantly trying to find new uses for cloud infrastructure beyond simple storage. Much of this has been focused on ways to reduce the processing power of local devices by offloading some computation to the cloud [1] [2]. Much of the research on cloud systems themselves have focused on increasing the efficiency of cloud storage [3]. Other research has looked into new applications such as storage of temporary files to speed-up low-end devices (like phones) [4]. With ever increasing internet speeds, it is becoming increasingly practical to store more data and programs that run on a computer onto the cloud, making the computer "leaner" in the process.

Another technology that has seen steady attention over recent years is the FPGA. An FPGA is a computer chip that can be programmed to act like almost any chip. At the cost of speed and power consumption, FPGAs offer greatly increased versatility over dedicated components. FPGAs have mostly seen application in specialized computing such as in industrial settings [5]; however, recent developments such as Linux support have been made to allow for FPGAs to be more readily utilized in general computing. One group of researchers, for example, were able to demonstrate a notable speed-up when using FPGAs for hardware acceleration of encryption/decryption algorithms [6].

Due to the rise in popularity of cloud-based devices, there has been growing interest into research that bridges the extensibility of the cloud with the reconfigurability of the FPGA. For example, there has been research into creating a new interface framework for an FPGA to communicate efficiently with a cloud-based computer [7]; however, there has not been any research into combining the cloud's storage abilities with the FPGA's reconfigurability to create a leaner

OS. There has been research in creating this lean OS by sacrificing some form of hardware or functionality - for example, reducing the memory bank size and making programs constantly load in and out of it even when running [8] - however, never at the expense of removing the hard drive outright.

Motivation

Our focus differs from these prior approaches: we use the cloud's remote storage capabilities to store not only data, but programs, the operating system, and device drivers as well. We leverage the increasing speeds of the cloud infrastructure to create what might be called an "on-demand" computer architecture. This system uses the cloud for all data storage (including the operating system). The only dedicated hardware on our computer is the main memory (RAM), the processor, and the peripherals for internet connection. Whenever any other hardware is needed (such as USB drivers or audio cards) the system downloads the information to implement the hardware on the FPGA, and the corresponding device driver loads into the processor.

Compared to prior work in this field, we leverage the advantages of both the cloud and FPGA to a greater extent. Our design allows for a lean, cost-effective system since hardware is only present when needed. This system also offers improved security since a fresh copy of the operating system would be downloaded at every boot; thus, malicious software from one boot would not persist after the computer is shutdown. This design helps demonstrate the power of both the cloud and FPGAs for cost-effective computing.

Overall Architecture

The architecture of the system consists of a cloud server that acts like a local drive, and a modified computer that is specially constructed to be lean. The modifications of this computer consist of a permanent network interface to communicate with the cloud server, a lean CPU that forgoes many typically functions like virtual memory via the removal of the MMU, and a reconfigurable FPGA that handles all peripheral interactions. The overall architecture is shown in **Figure 1** and each of its parts will be discussed in the following chapters.

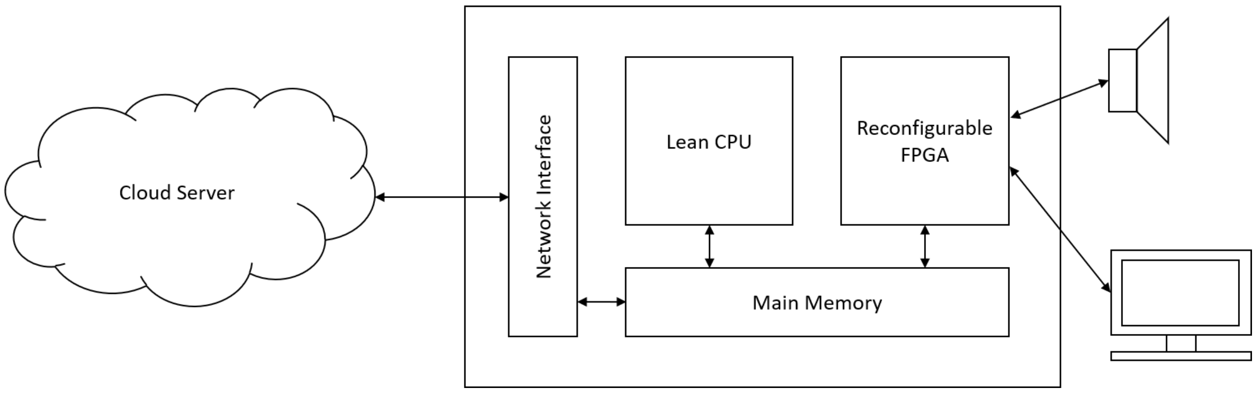

Figure 1: Overall planned architecture

CHAPTER II

CLOUD SERVER AND DATABASE

The cloud server acts as the mediator of communication between the database system and multiple clients (local computers) shown in **Figure 2**. To ensure that the database itself is internally consistent at all times, the database can only be called by the server and handle only one transaction from a client at a time following the ACID database properties. Following this logic, the server communicates with only one client at a time and reserves a backlog of transactions to perform after each action. The following sections will discuss the intricacies of the database construction, the server construction and the server-client interaction in greater detail.

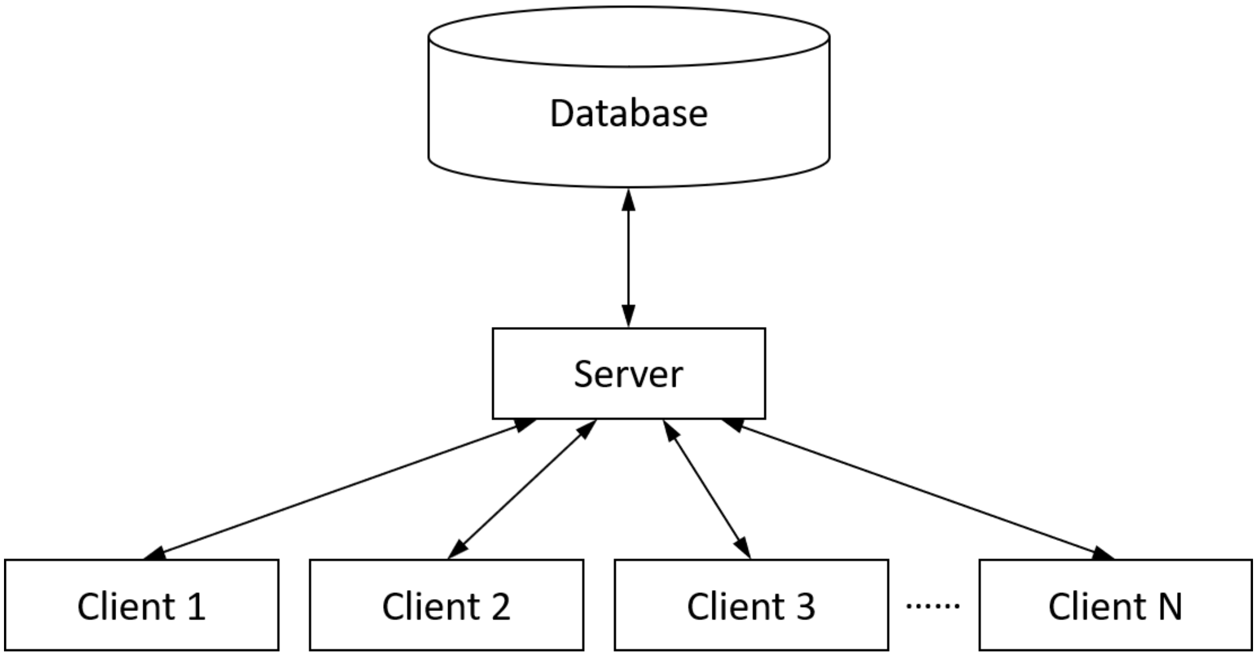

Figure 2: Higher-level view of network communications

Database Types

There are many types of databases each defined by their implementation and purpose. However, for the purpose of large-scale data storage which is essential for CODArc, there are three main types to consider: file-based, block-based and object-based.

File-based storage databases use file-systems to emulate the folder structure of a client and copy it onto the server. This allows the structure of the database itself to be easily understood and worked with at the cost of scalability due to the inherent nature of folder hierarchies.

Block-based storage databases use fixed-sized blocks to store data. Since each block only contains data and is referenced with a simple address, the server can easily find any piece of data in the database. This allows this type of database to have extremely low latency.

Finally, object-based storage databases use a type of container called an object to store data. Since objects only need a unique identifier and can hold a great amount of metadata, information can easily be queried in a large pool of data. This allows object-based databases to have remarkable scalability.

For CODArc, we to used a predominantly object-based database with some features of the other two. Since CODArc replaces the local hard drives of client computers and typical hard drives can contain thousands of files, scalability is of great concern. However, since we also want the speed and structure of block and file-based databases respectively, we have incorporated some of their features into the database.

Permission Handling

To facilitate proper data handling between various clients, permissions have to be defined properly for each object in our database. Assuming every object is owned by a user, each object contains two types of permissions: user permissions and group permissions. This structure is illustrated in **Figure 3**.

User permissions are implemented in a similar way to how Linux assigns user permissions for files - using a 3-bit number (0-7). The most significant bit pertains to read, the next bit to write and the least significant bit to execute. If a one is assigned to a bit location, then access is granted for that function; if zero is assigned, then access is denied.

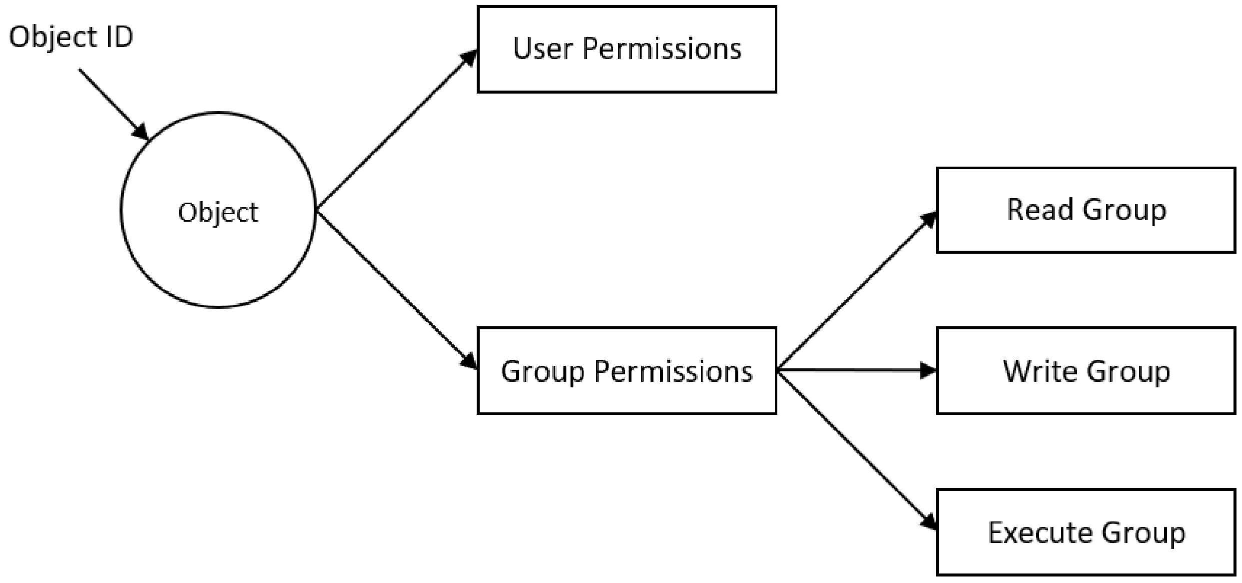

Figure 3: Permission structure

In contrast, group permissions are implemented differently to how Linux assigns them. Linux uses the same system as user permissions as with group permissions - a 3-bit number. Since our database needs to handle a large volume of users with potentially a large number of created groups, group permissions are split into three units called the read group, write group and execute group. Each of these units contain a group name which allows only that group to have that function. For example, if a group called ’Texas A&M’ was assigned the read group, then only members of ’Texas A&M’ can read that object. This helps facilitate a universal group called ’all’ which allows universal files, files that can be accessed by anyone, to exist. This in turn allows important programs like the OS or frequently used files like browsers to exist in one place where everyone can use them.

Database Implementation

The CODArc database handles all information regarding the location of objects, and the metadata associated with each object including permissions. To facilitate this, the database is split into a hierarchy of three levels. From lowest to highest, they are the objects, the user/group file tables, and finally the user/group list.

Every object in the database references a singular file on the cloud server and is unique to each user. Each object is defined by an OID and a direct path the owner uses to locate it. The OID is globally unique (among all users). The direct path is per-user unique: multiple users can have different files with the same direct path, but no user can have multiple files with the same direct path. This direct path is what the client sees and is necessary as an abstraction to emulate the folder structure of a normal computer. The object also contains the name of the server file containing the data, user/group permissions, and a set of metadata. With the exception of the direct path and metadata, which is found in a structure in the next paragraph, the information of each object is directly inserted into a dynamically allocated object structure. When a user deletes a local file on the client end, the server deletes the object on its end.

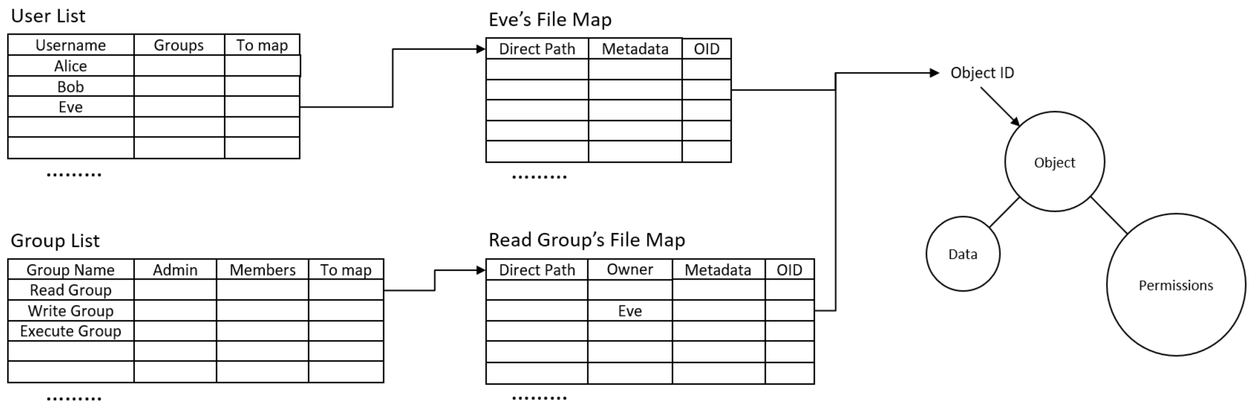

Figure 4: Database structure

Figure 4 illustrates the structure for storing direct paths and metadata. Each user and group has a file table – known as the individual and group map respectively – which catalogs all the objects that are available to them. To maintain speed, this structure is implemented using hash tables where the key is the local direct path, and the value is the location of the object and a set of metadata. The user and group file maps are identical with the sole exception of an additional attribute found in the group file map called ’owner’. As a rule, only the owner of the file can add group permissions to the file and so it is important to note the owner in the file map.

At the highest level there is a table containing a list of users called the user-list and table containing a list of groups called the group-list. Similar to the file tables, the user-list and group-list are implemented using hash tables with the key associated with the username/group name and the value listing the location of the file table and a set of associations. For the user-list, the associations are the groups the user is a member of, whereas for the group-list, the associations detail the members of the group. The group-list also lists the name of the administrator of the group who by virtue can add or remove members of the group at any time.

Searching the Database

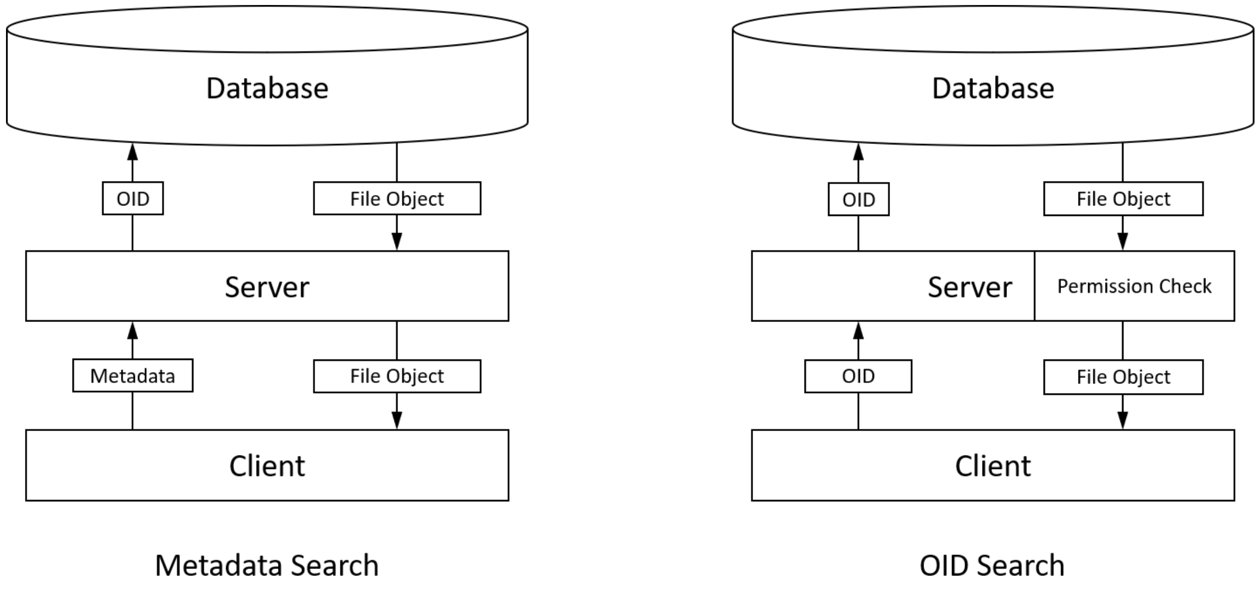

Figure 5: Search mechanisms

The CODArc database has a variety of functions that the server uses to fulfill the requests of the local client, such add/remove groups, and add/update/remove objects. Most of these functions are simple requiring only basic data updates. The search functions however, are a little more complex. To make sure that a client can find any data placed within the database that is accessible to them, three search mechanisms are available to the user: OID search, metadata search and direct-path search. **Figure 5** illustrates both methods.

OID search is the quickest type of search in this database. Since the database knows the location of all OIDs, the database can simply find the data associated with it and send out the object. However, the server needs to perform a permission check on the returned object because any user can request any OID at will. Once the server has verified that the permissions of the object matches the user, then the server can send the requested data to the client.

General metadata search is the slowest search mechanism offered by the database. The client sends a list of metadata to the server and the server queries the database for any matching data. However, since the database is stratified into user and group file tables, the database only needs to search for data within the matching user's file tables and any associated group file tables. Since the database searches through these tables, there is no need for a permission check and the server can immediately send out the object. If there are multiple objects that found during the query, the server will send a confirmation to the client informing them of such, and the client can choose to take one or all objects found.

Direct-path search is a specialized form of metadata search which is faster than the general kind. Since the file tables are constructed using hash tables, the database can easily perform a key search using the direct-path and obtain the object that way. The database will always search through the user map using this method and will only search through any subsequent group maps if the file was not initially found.

Server Implementation

As discussed in the beginning of the chapter, the server serves as the intermediary between the clients and database. As such, the server handles two main tasks: facilitating server-client communication (discussed in the next section) and updating the database. To ensure that the database uses the least amount of storage for files, the server is also responsible for ensuring that only one copy of a specific file is stored in the database in a process called deduplication. When the server takes in a new file, the server hashes it and checks to see if the resulting hash matches a hash that the server has seen before. If the server has, it checks the data of each file with the same hash against the new file, and if the data matches that of the new file, the new file does not get stored. This process allows for the minimum amount of files that are needed on the database, resulting in a more scalable and resource-efficient database.

Server-Client Communication

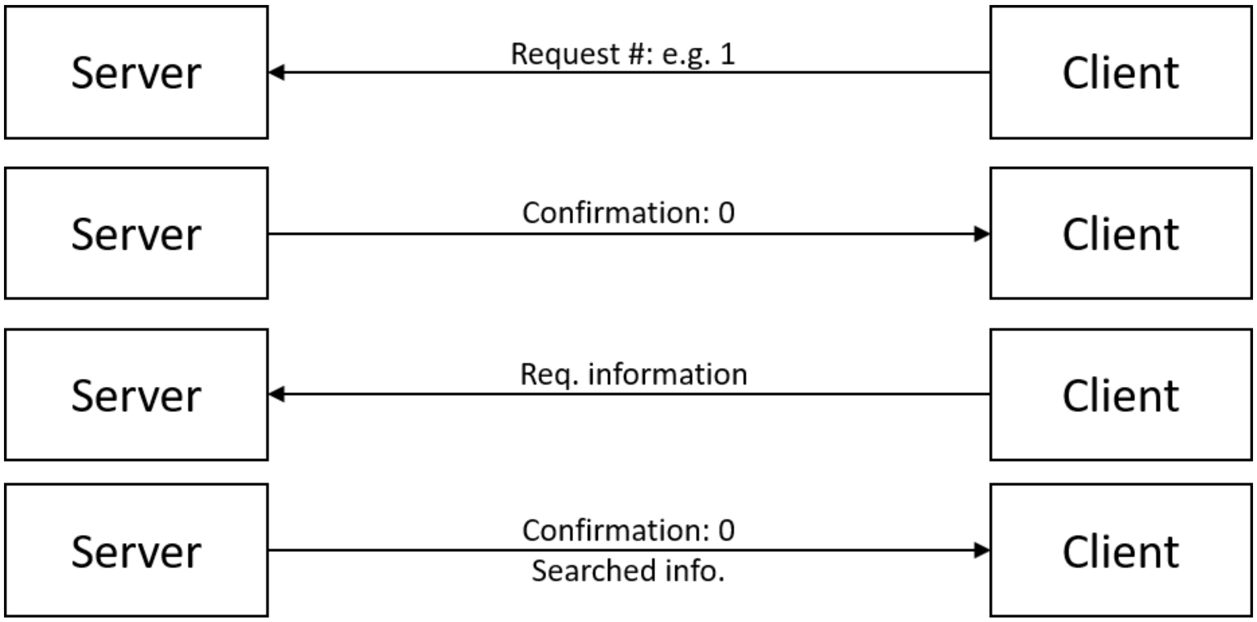

Figure 6: Server-client communication system

The communication system for how a client computer talks with the cloud server is based entirely on TCP/IP. The client will start the connection on start-up and until shutdown, it will exchange any local storage data with the cloud whenever the OS demands it. The basic structure for the client-server communication is as follows. When the client needs something from the server like storing new data or searching for a file, it will send a specific request number which details the type of request to the server. Once the server acknowledges and confirms the request, the client will send any required information to the server. Once the server completes the request, the server will send either a confirmation back to the client if the request updated server data, or files and search information if the request was a search. A summary of this process is shown in **Figure 6**.

CHAPTER III

SYSTEM SOFTWARE

Due to changes in the architectural hardware, changes in software are necessary to address them. The lack of virtual memory and subsequently the MMU that are typically commonplace in a computer, require an OS that adapts to these major modifications. Likewise, the lack of a local hard drive necessitate both a change in the OS and different boot software that communicate directly with the cloud to download data into main memory. The following sections will discuss the software in greater detail, as well as future work that is necessary to fully complete the system.

Operating System

In a standard OS, virtual memory is a vital part to how modern computers function. It provides the abstraction necessary for software processes to not only be isolated from one another but also assume that they each have use of unlimited memory space. However, an actual computer only has a certain amount of available RAM installed. To provide this abstraction, a translation between virtual memory and physical memory is necessary each time a program reads from or writes to memory. This is the job of the MMU, a hardware unit that handles all memory translations. In our project, in order to increase computation speeds and reduce the size of overall hardware, this piece of hardware is removed, which the OS has to account for.

The OS also has to account for the lack of a local drive. In a conventional computer, a disk drive is connected directly to the architecture and serves as the hard drive of the system. Since the newly created cloud server now takes over this function, the OS needs to account for this as well as shown in **Figure 7**.

To address the first problem, the lack of virtual memory, an OS that does not use this function was necessary. In our case, we found a variation of Linux called uClinux that aimed to supply processors that have no memory management. Although this OS was plagued with a host of limitations all drawing from the lack of memory management, it served as the keystone for the software found on the local computer.

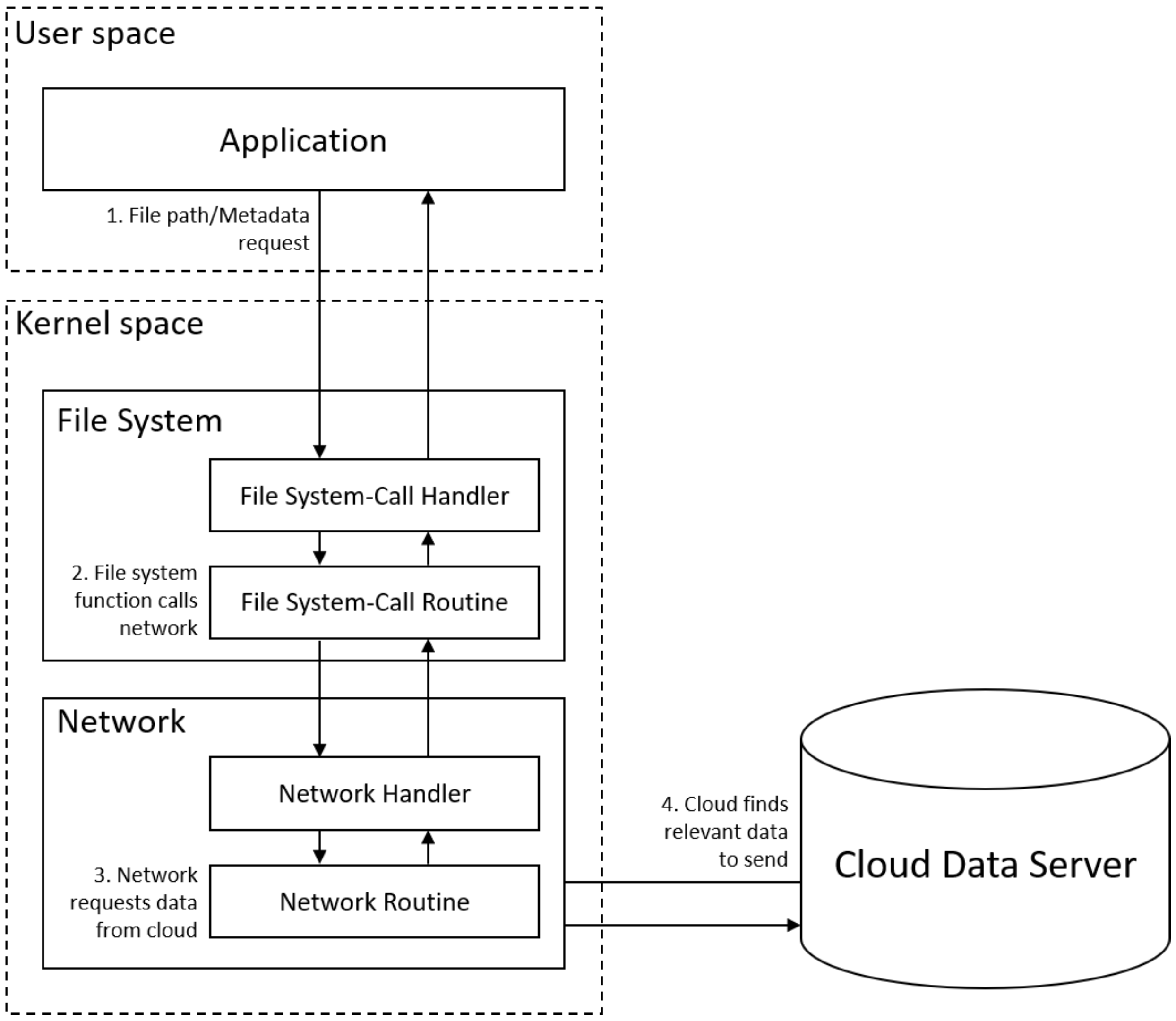

Figure 7: Overview of a networked file system

To address the second problem, we use an existing networked distributed file system called NFS, which was available as a file system option on uClinux. The only modification that was necessary to complete was modifying the cloud server to match the same NFS protocols that are expected when data is sent to the cloud using this method.

System Boot Software

The purpose of the system boot software is to load the OS into RAM for standard operation. This would be done by running an initial bootstrap program found in the BIOS. The program would run the standard diagnostics on the hardware and once complete would run the OS boot loader as shown in **Figure 8**. Typically this loader would load the OS from the local disk drive to main memory where the OS would take over. However, in our case, since our storage is found within the cloud, we needed a different solution for this loader.

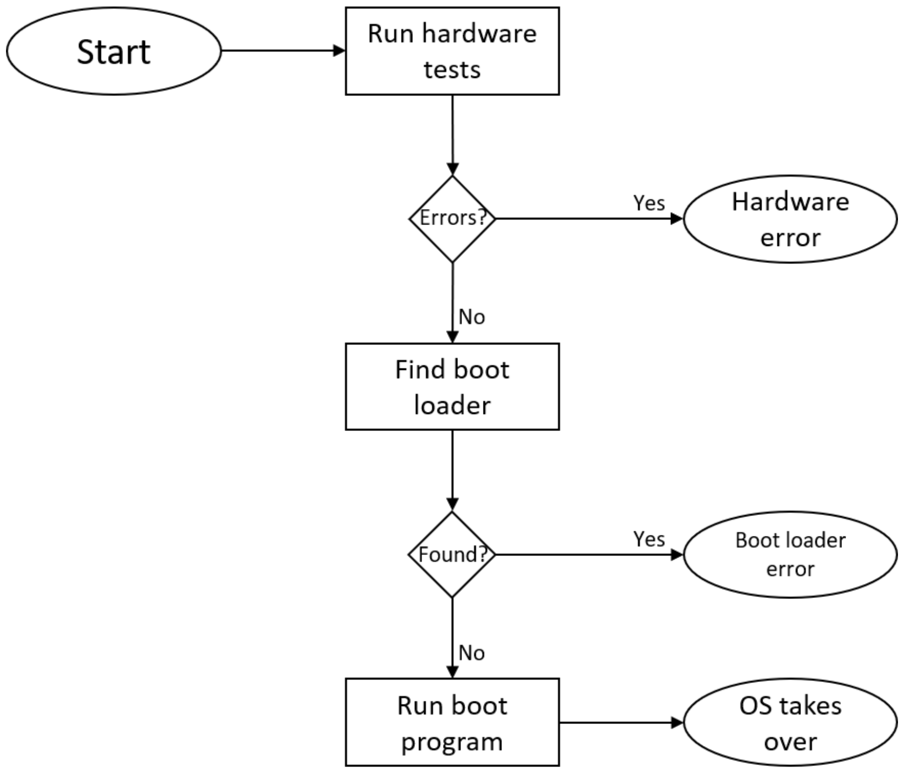

Figure 8: Boot sequence of a computer

We use an existing solution called network boot over TFTP to communicate with our server. This configuration allows us to use the same NFS protocols for booting as when the OS handles networked files, allowing the booting changes to the cloud server to be the same as the OS changes.

Future Work

As of writing this thesis, we are currently not fully finished with the implementation of the software section. Although we listed what we hope to accomplish in the two sections above, they remain incomplete.

The main piece of work that still needs to be done in regards to software is modifying the cloud server to accept the NFS communication calls from the client. Presently the server has a specialized format for communicating information between clients, however it is not the TFTP that an NFS client expects. So the server needs to change not only the protocol that it is currently using which is TCP/IP but also translate the functions that an NFS server requires into functions that the server can handle. Once this is complete, the entire server and client software need to be integration tested and finalized.

CHAPTER IV

HARDWARE STRUCTURE

While a finished version of this computer system would have dedicated hardware for each component, the prototyping phase addressed in this paper uses a single FPGA to develop the entire system. **Figure 9** illustrates a high-level version of this system. This chapter first aims to explain the structure of the hardware system, considerations for its design, and other concepts necessary to understand its operation. The SPI Flash Controller would not be discussed since it is only used as non-volatile memory in this design. Future designs might use other methods of non-volatile storage. The operation of the CODArc hardware system is explained in the final section of this chapter.

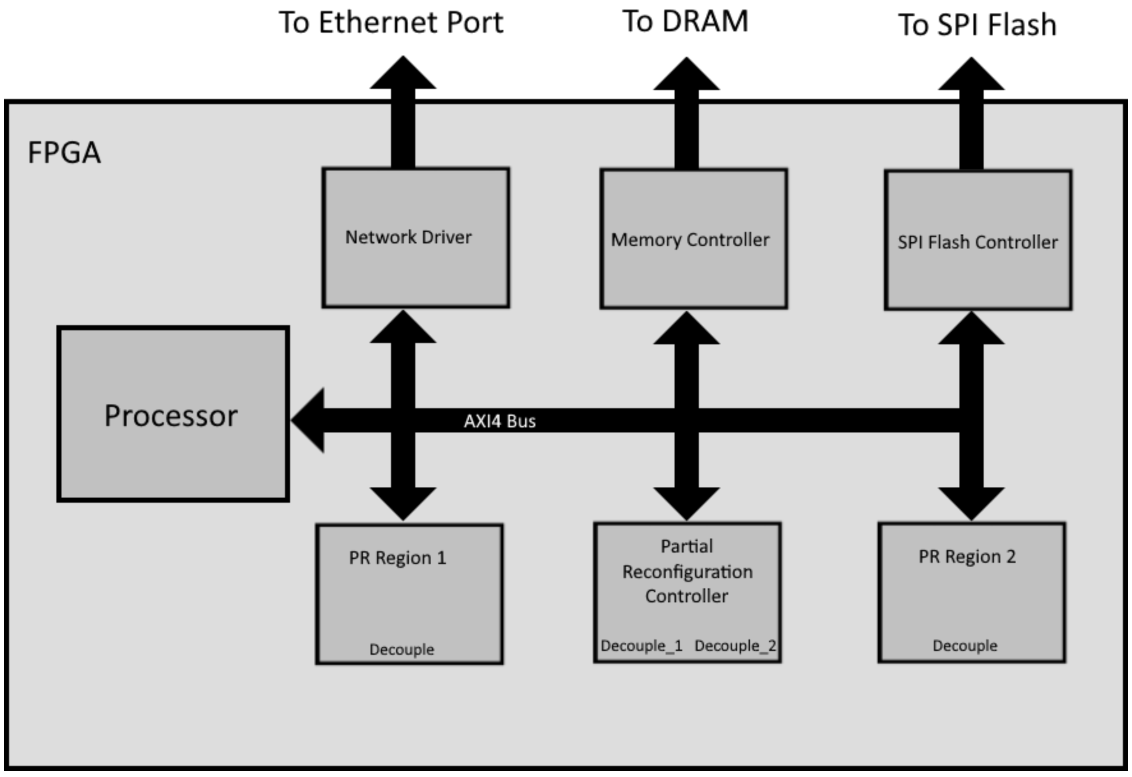

Figure 9: Diagram of the system on the FPGA

FPGA Hardware

A FPGA is a piece of hardware that can be programmed to act as any other hardware by configuring a vast amount of LUTs. The usage of an FPGA was chosen because it offers a method to test the design in hardware, while also being multiple orders of magnitude cheaper and simpler than designing and fabricating dedicated computer chips. An FPGA is also necessary for CODArc in order to support the on-demand-hardware aspect of the system. This factor will be discussed later.

The primary consideration for choosing an FPGA was the size of the chip. Each FPGA chip has a limited number of LUTs and other resources on-chip. Therefore, larger designs need larger FPGAs, which have more resources. For CODArc, an entire computer system has to be implemented, with plenty of room leftover for multiple partially-reconfigurable peripherals. This means the FPGA needs to be fairly large for a one-chip system.

The other factor under consideration was the choice in FPGA board. FPGA chips are very complicated. Consequently, for prototyping, they are often part of a board that includes hardware to configure and test the FPGA. These boards often provide interfaces (such as USB and Ethernet), as well as on-board memory. For this design, we needed a board that offered sufficient on-board memory to store the OS and other programs. We also needed the board to include an Ethernet connection port.

For this design, the Artix-7 FPGA by Xilinx was chosen. The Artix-7 offers support for dynamic partial reconfiguration (PR), and a large enough amount of LUTs to support the entire system. The Artix-7 is available on the Digilent Arty-A7 board. This board includes 256 MB of DRAM for program storage, and 16MB of flash memory for persistent storage. It also has an Ethernet connection. For programming and testing, the board can interface with an external computer through USB.

Partial Reconfiguration

One of the core pillars of this project is the usage of partial reconfiguration (PR) to change hardware peripherals during run-time. Standard FPGA configuration is performed by sending a file called a bitstream to the configuration port of the FPGA. To perform PR, a special "partial bitstream" is sent to the FPGA after the initial configuration is done. The Artix-7 is able to automatically recognize if a bitstream is partial or not.

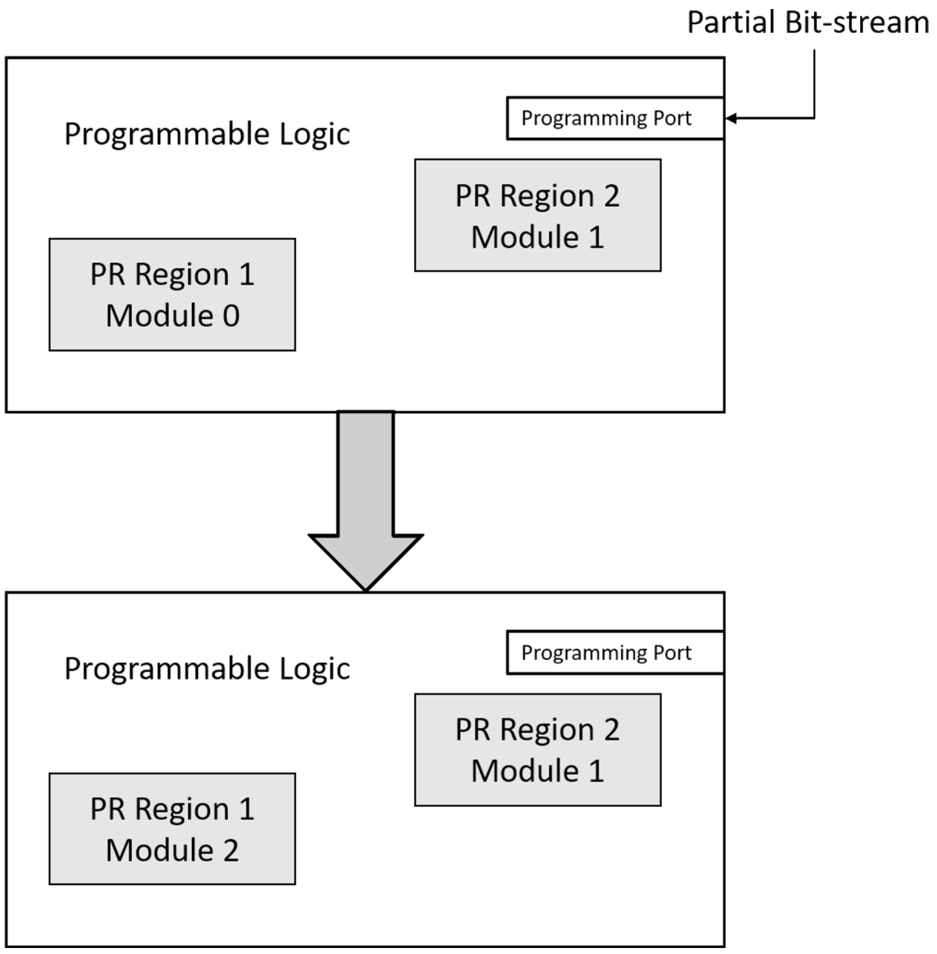

Figure 10: Diagram of partial reconfiguration

Figure 10 above shows the Programmable Logic of an FPGA with two PR regions – each region is configured with a module. PR regions are declared when creating the initial bitstream, and only the can be changed by PR. Each PR region is assigned a set of PR modules at the time of synthesis. Each module describes a configuration of the region. PR is able to substitute the module of one region for another module without affecting anything outside the region. As a result, there is one partial bitstream for every region-module combination.

Partial Reconfiguration Controller

The most common way to perform PR is to have an external system feed the partial bitstream to the programming port. In CODArc, such an external system would not be present. The partial reconfiguration controller (PRC) is a component implemented on the FPGA that allows the system to perform PR on itself. The PRC is placed on the main system bus so that software on the processor can execute PR.

Processor

The core of any computing system is the processor. Since the Artix-7 does not have a dedicated processor, a processor is implemented in programmable logic. When deciding what processor to use, a lot of weight was placed on the removal of the memory managment unit (MMU). The MMU is discussed later in this section.

The MicroBlaze soft processor, developed by Xilinx, was chosen for multiple reasons. First is availability: the MicroBlaze is built into the Vivado Design Suite (the program used to program the FPGA). It is widely used for applications on Artix and other Xilinx FPGAs, and there are online tutorials for common configurations. The number one reason that this processor was chosen is that it is highly configurable. Of the processors that were considered, it was the only one that could be synthesized without including a MMU (discussed later in this section).

Memory Management Unit

The MMU provides hardware support for virtual addressing. Virtual addressing gives programs the illusion that they have the entire computer memory to themselves (an explanation of how this is accomplished is outside the scope of this paper). This provides a wide array of benefits for programmers. The MMU also executes security checks to prevent programs from accessing restricted memory regions. This make it more difficult for malicious software to attack the system.

In CODArc, a fresh install of the system is available on every boot. Therefore, when designing CODArc, there was no concern whether a malicious program was given free reign, since the impact would not persist after reboot. Since the MMU adds extra steps to memory access, it slows the system down. As a result, we decided to list removal of the MMU in our specifications for CODArc. Theoretically, so long as software running on the system was specialized to operate without a MMU, this would speedup the system.

Interface to PR Regions

Since partial reconfiguration only changes the content of PR regions, all possible modules of a particular region must use the same interface to the rest of the system. CODArc requires that each PR region could become any driver, thererfore a robust interface between PR regions and the rest of the system is required. The AMBA AXI4 interface is the go-to bus interface for MicroBlaze processor systems – the detailed specifications of AXI4 are outside the scope of this paper. By using an AXI4 interface to communicate with the PR regions, the same protocol could be used to communicate with any module. Through AXI4, the processor could address, read, and write specific registers within the module. The regions are only designed with an AXI4 slave connection; therefore the processor must initiate all communication with the modules. To resolve this, the regions also have an interrupt bit as an output. Therefore, if any drivers need to communicate with the processor, they simply trigger an interrupt.

Operational Overview

This section aims to explain how the system would operate. This section will go into the details of starting the system, programming the peripherals with new bitstreams, and arbitration of I/O signals from peripherals.

Power-On

When powered on, an initial configuration file – stored in non-volatile memory – will program the FPGA with the CODArc hardware system. This configuration would initialze at least one of the PR modules as a USB driver so that the user could interface with the computer through keyboard and mouse. This configruation will also initialize the bootloader program into memory. Next, the latest version of the OS will be downloaded from the cloud database and placed into local memory. The remainder of the initialization process will be handled by the software.

Partial Reconfiguration

The PRC handles all partial reconfiguration; however, software follows a specific procedure to accomplish the task. First, the desired bitstream is downloaded from the cloud and placed in local memory. Next, the OS will decided which peripheral will be replaced. The PRC stores its configurations in registers within the controller; these registers are assigned addresses so that they can be accessed like any normal memory address. Depending on the peripheral that to replace, the processor will write the location and size of the new partial bitstream to the appropriate registers within the PRC.

Once the system is configured to perform partial reconfiguration with the new bitstream, a start signal is sent, telling the PRC to perform partial reconfiguration. During this process, a decouple signal is sent to the interface between the AXI4 bus and the module that is being changed. This signal takes the form of a wire being driven high. This signal will prevent any data from being transmitted along the bus to the module while PR is being performed. If the module was not decoupled, there is a risk that registers would be written to during configuration, and the FPGA might sustain damage. The decouple signal also prevents any external I/O signals from entering or leaving the module. Once partial reconfiguration is completed, the decouple wire is driven low, allowing the module to communicate with the rest of the system. Since PR changes the entire region, the decoupler is placed between to the PR region and the rest of the system.

I/O Arbitration

The design of CODArc intends that any PR region could be a peripheral that drives system I/O ports. For example, if there were two regions, and only one USB port, either region could be a USB driver. Partial reconfiguration can only modify one region at a time, and nothing outside the region could be reconfigured. Therefore, a standardized interface for I/O was created at the connection to each PR region. Regardless of the module, every region has a bi-directional port for every board I/O. This is referred to as the GPIO (General Purpose I/O) connection. Each bit of the GPIO line has matching "request" and "grant" ports that connect to an arbiter.

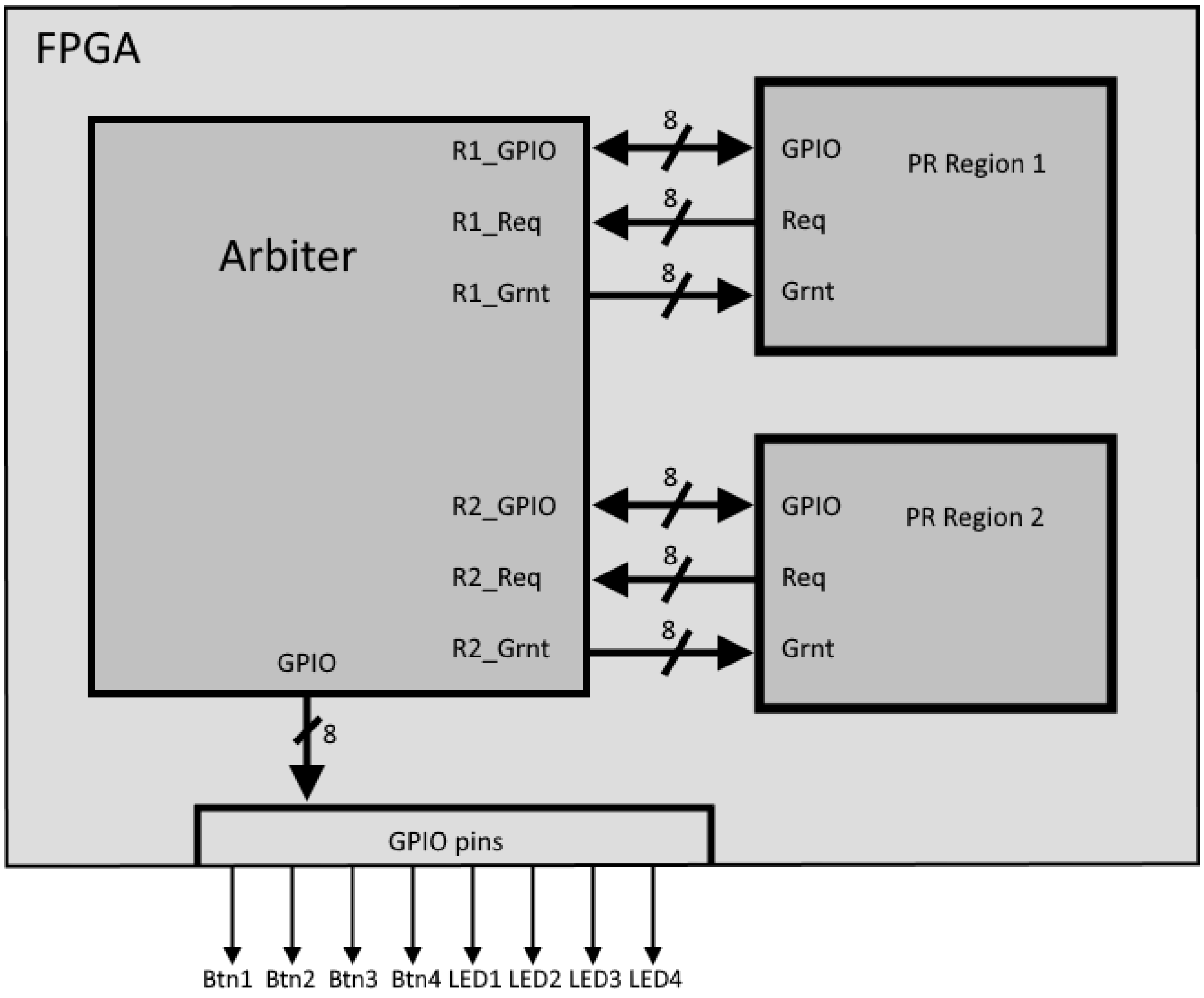

Figure 11: Example of arbitration system

The following is an example to explain how this system functions. Let the only I/O devices on the system be 4 buttons and 4 LEDs, thus a total of I/O 8 wires. This system is illustrated in **Figure 11** above. Region 1 is configured with module 1, which reads the value of the buttons, and illuminates the corresponding LED. Module 2 is configured into region 2. This module is a 4-bit counter that displays it's output on the LEDs. After initial configuration, region 1 will set all 8 bits of the request lines high, since module 1 requires all buttons and LEDs. Region 2 will request the bits associated with the LEDs. The Arbiter will see that both systems request all 8 GPIO pins. For each GPIO pin, whichever module makes the request first is given permission to use the pin (region 1 is given priority if they tie). The Arbiter informs the module of its decision by setting the associated grant line to 1. The arbiter then routes the signals between the GPIO pin and the chosen region's GPIO port.

Next, assume region 2 was given control of the I/O ports it wants. This means that region 1 is given access to the buttons, but cannot output their values to the LEDs. Next, region 2 is reconfigured as module 1. During PR, the decouple signal sets region 2's GPIO port to high impedance, and request to low. This causes the arbiter to grant control to region 1. Once PR is completed, region 2 will not get access to the ports until region 1 gives up control.

Since all regions can independently request any GPIO connection, this system allows any driver to be placed in any region. In future designs of CODArc, this arbiter could be enhanced to offer a more robust arbitration method. One possible improvement is to add a "priority" channel to each region. This channel would read the value of a software-writable register that assigns priority values to each region. If multiple regions request a GPIO pin, the one with the higher priority is given control. This modification would allow the OS to directly decide which regions controls which ports.

CHAPTER V

CONCLUSION

The concept of a computer that has its local drive on the cloud is not entirely new. However, we planned to innovate on this by having the computer be leaner than any computer that has come before it. To achieve this we removed fairly essential portions of the computer like the MMU and configured a piece of reconfigurable hardware to switch peripherals in and out when not in use all in an attempt to have a lean computer.

Unfortunately as of writing this thesis, this project is not fully complete. The scale of this project was highly ambitious as it required the development of a functional cloud database, a custom operating system, and complete computer system. The removal of typically essential hardware also made the project much more difficult. While much of the high-level design was completed as well as the implementation of a complete data server, many of the intended deliverables were never completed such as the local software and hardware. We hope that someone will be able to use the information we gathered here to complete this project in the near future.

REFERENCES

[1] D. G. Chandra and D. B. Malaya, “A study on cloud os,” in *2012 International Conference on Communication Systems and Network Technologies*, pp. 692–697, May 2012.

[2] G. Xiong, T. Ji, X. Zhang, F. Zhu, and W. Liu, “Cloud operating system for industrial application,” in *2015 IEEE International Conference on Service Operations And Logistics, And Informatics (SOLI)*, pp. 43–48, Nov 2015.

[3] S. V. Pius and S. Suresh, “A novel algorithm of load balancing in distributed file system for cloud,” in *2015 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, pp. 1–4, March 2015.

[4] D. Chae, J. Kim, Y. Kim, J. Kim, K. Chang, S. Suh, and H. Lee, “Cloudswap: A cloud-assisted swap mechanism for mobile devices,” in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 462–472, May 2016.

[5] D. Meyer, M. Eckert, J. Haase, and B. Klauer, “Generic operating-system support for fpgas,” in *2016 International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*, pp. 7–12, May 2016.

[6] U. Langenbach, S. Wiehler, and E. Schubert, “Evaluation of a declarative linux kernel fpga manager for dynamic partial reconfiguration,” in *2017 International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*, pp. 13–18, May 2017.

[7] J. Mbongue, F. Hategekimana, D. Tchuinkou Kwadjo, D. Andrews, and C. Bobda, “Fpgavirt: A novel virtualization framework for fpgas in the cloud,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 862–865, July 2018.

[8] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, Nov 2004.