LIVE MIGRATION OF FPGA APPLICATIONS

by

Daniel Ly-Ma

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical & Computer Engineering
University of Toronto

© Copyright 2019 by Daniel Ly-Ma
Abstract

Live Migration of FPGA Applications

Daniel Ly-Ma

Master of Applied Science
Graduate Department of Electrical & Computer Engineering
University of Toronto
2019

With the recent and growing trend of Field Programmable Gate Arrays (FPGAs) being deployed into the data centers, cloud computing service providers are finding it difficult to manage these devices efficiently because traditional server management concepts and techniques are not yet available for FPGAs. In this thesis, we explore how to bring one of these management techniques, live migration, to FPGAs. We develop a system using partial reconfiguration, bitstream readback and memory pre-copying to live migrate a running hardware application from one physical FPGA to another physical FPGA. Our system is transparent and does not need to modify the hardware application, allowing for hardware designed with RTL and/or HLS to be migrated. Our approach can migrate hardware applications that use off-chip memory and that are network connected. We demonstrate this by live migrating a hardware Memcached server.
Acknowledgements

I would like to thank my supervisor, Paul Chow, for his invaluable support and guidance throughout these last few years. Thank you for allowing me to develop and pursue my own ideas and for somehow always finding the time to talk and answer my questions during those countless impromptu drop-ins by your office :-). This work would not have been possible without your unending patience and encouragement. I would also like to thank my second supervisor, Andreas Moshovos, for his support and exceptional patience and understanding during these past few weeks.

To my mother, May-Kim, and sister, Karen, for their unconditional love and support. Thank you for keeping me entwined with your daily lives and making me feel at home even though I’m so far away from you.

To the many friends and colleagues that I have met during my time here at the University of Toronto, especially Daniel Rozhko, Roberto DiCecco, Charles Lo, Varun Sharma, Naif Tarafdar, Nariman Eskandari, Qianfeng (Clark) Shen, Geoffrey Elliott, Justin Tai, Zhiqiang Liu, Eric Fukuda, Thomas Lin, Fernando Martin Del Campo, and Camilo Vega. Thank you for all your help and support and for making graduate school fun.

To Jaro and the ECE technical support staff, for your prompt responses and spot-on diagnosis of various technical issues that I have had while working on my thesis.
# Contents

1 Introduction ................................. 1
   1.1 Contributions ............................ 3
   1.2 Overview ................................ 4

2 Background ................................. 5
   2.1 Field Programmable Gate Arrays ............ 5
      2.1.1 Partial Reconfiguration ............... 5
      2.1.2 Decoupling ........................... 6
      2.1.3 Configuration Memory Readback and Restore . 7
      2.1.4 AMBA AXI Protocol ..................... 9
   2.2 Networking Concepts ...................... 11
   2.3 Live Migration of Virtual Machines .......... 12
   2.4 Memcached ................................ 13
   2.5 Related Work ............................. 14

3 System Design ............................. 16
   3.1 Goals and Assumptions ..................... 16
   3.2 System Overview .......................... 17
      3.2.1 Minimizing Downtime ................. 19
   3.3 Hardware Design .......................... 21
      3.3.1 AXI-Stream Decoupling .............. 21
3.3.2 AXI Decoupling .......................................................... 22
3.3.3 PR Controller ............................................................ 24
3.3.4 Bitstream Patching ...................................................... 26
3.3.5 Write Snooper ........................................................... 27
3.3.6 Dirty Bit Table .......................................................... 28
3.3.7 Migration Engine ...................................................... 28
3.3.8 Clocking ................................................................. 31
3.3.9 TCP/IP Network Stack ............................................... 31
3.3.10 ARP Generator ....................................................... 32
3.3.11 Controller .............................................................. 32

4 Evaluation ................................................................. 34
  4.1 Functionality .............................................................. 34
  4.2 Test Setup ............................................................... 36
  4.3 Results ..................................................................... 37
    4.3.1 Resource Usage .................................................... 38
    4.3.2 Migration Performance ........................................ 41
  4.4 FPGA Features for Live Migration .................................. 47

5 Conclusion .................................................................. 50
  5.1 Future Work ............................................................. 51

Bibliography ................................................................. 53
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Application utilization of the PR Region</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Total utilization report</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Migration infrastructure utilization breakdown</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Memtier_benchmark configuration</td>
<td>40</td>
</tr>
</tbody>
</table>
List of Figures

1.1 A high-level view of our live migration system. .......................... 3

2.1 Different designs can be loaded into a PR region [14]. ................. 6
2.2 An AXI Interconnect. [18] ....................................................... 10

3.1 Block diagram of our system. .................................................. 18
3.2 Multiplexing circuits for decoupling handshaking signals. ............. 22
3.3 Counters for tracking AXI transactions. ................................... 24
3.4 A typical layout of a Xilinx FPGA bitstream [32]. ....................... 25
3.5 The Migration Engine header. ............................................... 29
3.6 Sequence diagram between the Migration Engines. ..................... 30
3.7 The MicroBlaze controller. ..................................................... 33

4.1 Test setup. ............................................................................. 37
4.2 Placement on the FPGA. ....................................................... 38
4.3 Impact of migration on the performance of the Memcached server. .... 42
4.4 Memcached live migration downtime. ..................................... 45
4.5 Dirtying rate of the synthetic application. .................................. 47
Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) are becoming more ubiquitous in data centers and on cloud computing platforms. Through their Project Catapult, Microsoft has demonstrated the potential of FPGAs in the data center by accelerating web search [1, 2] and machine learning [3] workloads. Since then, several cloud computing services including Amazon AWS, Alibaba Cloud, Baidu, Huawei, Tencent Cloud and Nimbix have started offering customers access to FPGAs on their cloud computing platforms [4].

Management of FPGAs in the data center is an ongoing challenge and is currently a highly active area of research [5, 6, 7, 8]. One way of managing FPGAs is to follow the approach that is done with traditional server hardware. Instead of providing the barebone server to a customer, they are given access to the server’s physical resources by way of virtualization with virtual machines (VM). Several VMs can run on a single physical machine, allowing for more efficient use of the physical resources of a server. Analogously, instead of providing customers direct access to an entire FPGA, they are provided with a portion of the device through virtualization of the FPGA. Research is ongoing to tackle the challenges of FPGA virtualization, such as how to accomplish multi-tenancy on an FPGA, while providing performance isolation and ensuring that clients are securely sandboxed from each other [5, 9, 10, 11].
We anticipate that once FPGAs become virtualized in the data center, other management features will be required to manage these virtual FPGAs (vFPGAs). One such feature that we anticipate will be needed is live migration. Live migration is a feature used for VM management; it allows a running VM to be moved from one host server to another host server without having to shut down the VM. The VM’s state, such as physical memory and network resources and connections, are migrated to the destination server. This process is mostly transparent to the VM, as if the VM had never moved. Live migration allows service providers to bring a server down for maintenance, mitigate effects of hardware failures and to perform load balancing without disrupting their users. We believe that live migration will be required for the same reasons for FPGAs as well because we know, from Putnam et al. in [1], that FPGAs can fail in the data center. They recover from FPGA failures by programming backup FPGAs and remapping their network to these new FPGAs, but it appears that they lose any work and data that was on the failed FPGA. With live migration, work and data could potentially be moved to a different FPGA before the failure occurs, minimizing the disruptiveness of an FPGA failure.

In this thesis, we investigate what it means to live migrate an FPGA application, and to see if the features in current FPGA hardware can be leveraged to perform live migration. Based on our findings, we also want to look into what other features might be useful to add to FPGAs to make live migration possible or more streamlined. We then propose a system to perform live migration of hardware applications between physical FPGAs that is less restrictive on the type of hardware application it supports compared to prior work. By using available features in some current FPGAs, such as bitstream readback and partial reconfiguration, and borrowing ideas from VM live migration, we propose a system that can potentially be used to live migrate most hardware designs. Our proposed system can migrate a hardware application’s on-chip state, along with other external resources such as off-chip memory and network connections. Figure 1.1 shows a
high-level view our system where we live migrate a hardware server application and its network connection to its clients from one physical FPGA to another physical FPGA. We demonstrate the potential of our system by attempting to live migrate a hardware Memcached server [12, 13].

1.1 Contributions

The main contributions of this thesis are as follows:

- The development of a flexible system that can potentially live migrate most hardware designs transparently requiring modification to these designs;

- The development of a system that could allow for migration of TCP/IP network connections;
• An evaluation of the effectiveness of pre-copying as a technique to reduce downtime of FPGA applications when performing live migration;

• An evaluation of the effect of using bitstream readback and restore as a state saving technique to perform live migration of FPGA applications on downtime;

• An investigation into impact on migration time when using bitstream readback as a state saving mechanism;

• A discussion on potential features to add to FPGAs to make live migration more feasible.

1.2 Overview

The rest of this thesis will be organized as follows. Chapter 2 provides background information for concepts that will be covered in later chapters and a summary of related work. Chapter 3 describes our proposed live migration system, starting with our goals and assumptions, followed by the overall live migration process, which is then accompanied by details on the hardware design. Chapter 4 describes our test setup and presents the results of our live migration system. Chapter 5 concludes this thesis and discusses potential future work.
Chapter 2

Background

This chapter provides background information for concepts and terminology that will be mentioned throughout the other sections of this thesis.

2.1 Field Programmable Gate Arrays

At a high level, an FPGA is a reconfigurable integrated circuit that is used for developing and prototyping large complex digital systems. It mainly consists of a large heterogeneous grid of logic, memory, and digital signal processing (DSP) blocks connected together through a reconfigurable interconnect. FPGAs can also have high speed IOs that allow for access to high speed interfaces, such as 100G Ethernet, DDR4 memory, and PCIe. More recent FPGAs come with multi-core ARM processors that allow for complex system-on-chip (SOC) designs.

2.1.1 Partial Reconfiguration

Partial Reconfiguration (PR) is a feature in some FPGAs that allow for dynamic reconfiguration of a section of the FPGA fabric at run time without affecting the other surrounding logic regions. The dynamic reconfigurable region is known as the PR region.
and the part of the FPGA design that does not change is known as the *static* region. Signals from the PR and static regions can interact with each other through interface pins that are found at the boundary between the two regions. Figure 2.1 shows a typical use case for partial reconfiguration where multiple hardware modules are built for a PR region and are swapped in and out of the PR region during runtime. This provides design flexibility by allowing parts of the FPGA design to change at runtime. Without PR, a change in the design would require a recompilation of the hardware and would require the FPGA to be taken offline for programming.

Several bitstreams are generated when using partial reconfiguration. A bitstream is generated for the entire design and is used to load the static portion of the design onto the FPGA. Smaller bitstreams called partial bitstreams, such as those shown in Figure 2.1, are generated for the design that fits in the PR region. The partial bitstreams are used to reconfigure the PR region during runtime.

### 2.1.2 Decoupling

Interface signals that cross between the PR and static region can be driven into an unknown state during partial reconfiguration that can lead to undesirable behaviours such as data corruption and bus deadlock. Designs using PR can avoid this by adding decou-
pling units to safely disconnect the PR region from the static region. A basic decoupler unit, usually implemented with a 2-to-1 multiplexer, drives a signal that is crossing between the PR region to a known stable value. Not every signal requires decoupling; usually control signals, such as handshakes and enables, are decoupled because they can cause unintended state changes and/or data being registered if they are left floating. A basic decoupler is adequate for simple communication protocols where transactions are completed in a single cycle. It is inadequate for more advanced interfaces where transactions can happen over multiple cycles because a basic decoupler does not have any concept of a transaction and will decouple a signal immediately when requested to do so; decoupling during the middle of a transaction can cause deadlock. A more advanced decoupler that can wait for a transaction to complete before decoupling is required to decouple these interfaces safely. Our design makes use of these multi-cycle transaction interfaces and requires us to design more advanced decouplers for these interfaces. We describe them in detail in Section 3.3.1 and Section 3.3.2.

2.1.3 Configuration Memory Readback and Restore

FPGA configuration methods depend on the FPGA vendor and architecture, so the methods and concepts described in the following sections are specific for the 7 Series generation of Xilinx FPGAs and may not completely apply to older and newer generations of Xilinx FPGAs. We mention this because we began this work using the newer Xilinx UltraScale generation of FPGAs and discovered that the PR architecture had changed enough compared to the 7 Series that the readback technique that we are about to describe was no longer compatible. We discuss this in more detail in Section 4.4. We target Xilinx FPGAs because Xilinx FPGAs have had the capabilities to read back configuration data since the original Virtex devices [15], where Intel has just begun supporting this feature in their FPGAs with their latest Stratix 10 devices [16].

FPGAs are programmed using a bitstream file. A bitstream consists of two com-
ponents: configuration instructions and configuration data. Several interfaces can be used to load the bitstream into the FPGA. These include external interfaces like JTAG, SelectMAP, and SPI and an internal interface called the Internal Capture Access Port (ICAP) for user logic to access the configuration space \[17\]. The configuration instructions direct the FPGA where to place the configuration data in its configuration memory. Configuration data contains information on how to configure the resources, e.g., configurable logic blocks (CLBs), memories, interconnect, etc. Configuration memory is organized into addressable units called configuration frames which consists of a column of the same resource, such as CLBs, BRAMs, and DSPs. A configuration frame is the smallest unit that the configuration interface can address; the size of a configuration frame is architecture dependent. It is 101 words (32-bits) for the 7 Series architecture \[17\].

Readback is the opposite of configuration where the configuration memory on the FPGA is read back through one of the configuration interfaces. Unlike a bitstream, readback data only contains configuration data and does not contain configuration instructions. It is designed to be used for verifying the integrity of the configuration memory against the original bitstream, which is required in environments such as in space where the FPGA’s configuration memory can easily be corrupted by radiation. However, for our system, we use readback to take a snapshot of the FPGA’s state for migration.

The FPGA’s registers cannot be written to or read back directly because they are not part of the configuration memory. As a result, each register has a specialized configuration cell that is used to load in initial values during configuration. After configuration, the values in these memory cells are copied into the registers. The 7 Series has a feature that enables us to capture the current state of registers into configuration memory. A capture operation loads the current state of the registers into their initialization memory cells. Since these initialization cells are part of configuration memory we can read them back during a readback operation. Only the values in the registers found in CLBs and I/O blocks can be captured. The values of internal registers that are found in DSPs
and BRAMs cannot be captured because they cannot be initialized and do not have initialization memory cells. Unlike the registers, LUT RAM and BRAM are part of configuration memory and do not require specialized initialization memory cells. They can be written to directly during configuration, and read back directly during readback.

In theory, the configuration is restored by programming the FPGA with the readback data and then pulsing the global set/reset (GSR) signal to load the initialized data into the registers and BRAM. However, from our experience, programming the readback data directly back into the FPGA only restores the state of the registers; the BRAM and LUT RAM values are not restored. The readback data seems to be missing control data to instruct the FPGA to initialize BRAM and LUT RAM values from configuration memory. We describe how we worked around this problem in Section 3.3.4.

2.1.4 AMBA AXI Protocol

The Advanced eXtensible Interface (AXI) protocol is a specification [18] by ARM for several bus interfaces. There are three main bus types outlined in the specification: AXI, AXI-Lite and AXI-Stream. AXI and AXI-Lite are meant for memory mapped applications with the main difference being that AXI has support for burst operations making it suitable for high performance applications while the AXI-Lite is more geared to accessing registers. AXI-Stream is a streaming specification and more suitable for streaming and data flow designs. There are several generations of the AXI specification. When we refer to the AXI protocol in this thesis, we will be referring to the fourth generation of the specification, AXI4.

Xilinx uses the AXI protocol in many of their intellectual property (IP) blocks including their networking and memory controllers. Our design targets Xilinx FPGAs and CAD tools, so we must design, mainly the decouplers and memory write tracker, around these protocols.
AXI

AXI is a high performance memory mapped bus that supports several different configurations for connecting AXI master devices to AXI slave devices, ranging from connecting one master to one slave to having several masters connect to several slaves on the same bus as shown in Figure 2.2. These connections are managed by an AXI Interconnect IP. A typical configuration and use case for AXI is for multiple masters to share access to off-chip memory by being connected to a single memory controller acting as a slave.

The AXI bus contains five separate channels: read address (AR), read data (R), write address (AW), write data (W) and write response (B). The AR and AW channels are for issuing read and write requests to the bus, respectively. The R and W channels are for transferring the requested data from the slave and to the slave, respectively. The B channel lets the slave notify the master if the last write transfer from the master has completed successfully. Handshaking between the slave and master ensure transactions on the channels are acknowledged by both parties.

The design for an AXI decoupler will have to take into consideration the relationships between the channels to effectively keep track of ongoing transactions and to prevent new ones from being issued.
AXI-Stream

AXI-Stream (AXIS) is a data streaming interface that is used in dataflow applications. Data flows in one direction for each AXI-Stream channel. A channel must consist of a data signal and handshaking signals and may contain several sideband signals as well. The sideband channels allow for additional information to be accompanied by the data. The sideband signal that is of most interest to us is the TLAST signal, which allows for the sender to mark packet boundaries. We will make use of TLAST in the design of our AXIS decoupler to ensure that we do not decouple in the middle of a transmission of a packet.

2.2 Networking Concepts

In our work, we transfer state information over the network between the host and destination FPGAs. We make use of the Transmission Control Protocol (TCP) protocol, by way of an open source hardware 10G TCP/IP network stack [19][20], to ensure that we have a reliable and ordered connection between the two devices, i.e., any data sent from the host is guaranteed to be received by the destination and the data is guaranteed to be received in the order it was sent by the destination.

Our hardware application, a Memcached server [12][13], also makes use of TCP with its own copy of the 10G TCP/IP network stack. Our migration system can move and keep the application’s TCP/IP connections open after migration by making use of the Address Resolution Protocol (ARP) protocol [21]. ARP is mainly used on a switched local area network (LAN) to discover which media access control (MAC) address owns a specific IP address. Each device connected on the network has a unique MAC address that can be used by a Layer 2 network switch to direct packets destined for a specific MAC address to the port where the device is connected to. ARP has a feature called gratuitous ARP that enables a device to notify other devices, through a broadcast to the
network, its MAC address and IP address. A network switch learns which port the MAC
address of a device is connected to by keeping track of which port it last saw an outgoing
packet tagged with the MAC address. By issuing a gratuitous ARP, we can tell the
network switch to update its table and redirect all packets destined for the application’s
MAC address to the destination FPGA’s switch port.

2.3 Live Migration of Virtual Machines

Virtual Machines are used by cloud computing platforms like Microsoft Azure [22], Amaz-
on EC2 [23] and Google Compute Engine [24] to manage customer access to their phys-
ical compute resources. A virtual machine is an environment created in software that
resembles the physical hardware of a computer. Software, usually an operating system,
can run inside this environment. Multiple VMs can run simultaneously on a single phys-
ical host. The hardware abstraction provides a layer of security by preventing a VM
from gaining direct access to the underlining physical hardware, essentially forming a
sandbox around the VM, preventing it from tampering with other VM instances running
on the same host. VMs are managed by a software called a hypervisor. A hypervisor
creates and maintains the virtual environment for the OS to run in, and it manages the
allocation of physical resources, e.g., RAM, CPU cores, storage, network source, on the
host machine to the VMs. Hypervisors come in two variants, Type 1 and Type 2. A
Type 1 hypervisor runs directly on the host hardware while a Type 2 hypervisor runs on
top of an operating system.

Live migration is the process of moving a running VM from one host machine to
another host machine without having to shut it down first. The VM is usually unaware
of this happening since all of its state, i.e., memory, storage and network connections,
are also moved as well. Microsoft Azure [25] and Google Compute Engine [26] use live
migration for several reasons:
1. To allow a physical host or other support infrastructure to be brought down for maintenance;

2. To move VMs off of failing hardware;

3. Load balancing a host in case it becomes over provisioned, e.g., there are not enough physical resources to service all the VMs.

The concept of live migration was first introduced by Clark et al. in [27] where they demonstrated live migration of a VM running on a modified Xen hypervisor. They minimized downtime of the VM through iterative transfers of the VM’s state over to the destination server while the VM continued to run on the source server, achieving down times as low as 60ms. They assumed the source and destination servers were connected together with a local area network (LAN). This allowed for issuing of a gratuitous ARP on the destination server to redirect all network traffic destined for the VM’s MAC address from the source server’s switch port to the destination server’s switch port.

2.4 Memcached

Memcached [12] is a caching server that is generally used for accelerating access to databases. At a high-level, it is a large in-memory hash table where data is stored and retrieved using a key-value pair. We demonstrate the capabilities of our system using an open source hardware Memcached server [28, 13] for two main reasons. Firstly, Memcached is a typical workload that is run in data centers [24, 29], allowing us to demonstrate a more realistic use case for our system. Secondly, Memcached requires access to both off-chip memory and to an Ethernet network, allowing us to more effectively demonstrate the capabilities of our system.
2.5 Related Work

There have been few works exploring live migration of FPGA applications. This is mostly due to the recent rise of FPGAs in the data center, so we expect more work to come out in the future.

The recent work by Vaishnav et al. in [30] proposes a versatile live migration technique for OpenCL Accelerators on FPGAs. They take advantage of the OpenCL execution model that allows for workgroups to complete in any order and ensures data is written back to off-chip memory at the end of workgroup execution. This provides a convenient point to perform live migration because all state information is stored in off-chip memory, discarding the need to save any state on the FPGA. Without the need to save and restore FPGA internal state information, they are able to rapidly perform live migration, resulting in very little downtime of the accelerator. Since workgroups can execute in any order, this allows them to perform load balancing by scaling up and down the number of workgroups running in parallel. Though this is a powerful migration technique, it is not very flexible since it only works for OpenCL accelerators or any other model where there are clear periods when the state is entirely in the memory, which limits the kind of hardware designs that can be used in such a system.

In [11], Knodel et al. proposed a system for migrating virtual FPGAs where they save the state of a vFPGA using bitstream readback. They have a unique feature that can extract and map individual flip-flops and BRAM configuration memory cells to different partial bitstreams of the same design, allowing them to restore designs to different sized PR regions. Their system can migrate designs to a different physical FPGA over PCIe. Their main design goal was to better manage and utilize the FPGA’s resources, while ours is to support different types of hardware designs and to minimize downtime.

There have been several works exploring multitasking and hardware preemption on FPGAs, a concept that is closely related to live migration because hardware preemption is needed to pause the hardware application for it be migrated. Proposed solutions to
hardware preemption fall into two categories, those that require modifications to the hardware and those that require no modifications. The no modification approach uses a technique called readback, to read back a hardware application’s configuration memory at runtime [11, 31, 32]. The technique is transparent to the hardware application, but it is not efficient since it does not differentiate between useful and useless state information, requiring the readback of the entire hardware bitstream. To reduce the state information that is needed to be saved, the hardware application can be modified using techniques such as embedding scan chains [33, 34] to extract only useful state information, and adding HLS checkpoint states [35] to bring the hardware designed with HLS to a safe state for preemption. As shown by Wicaksana et al. in [36], a useful property of HLS checkpointing is that it can be portable across different FPGA devices because checkpointing happens at a higher abstraction level. Though these techniques can be faster and more portable, they are less flexible for the end user because they require access and modifications to their designs, potentially resulting in higher resource utilization and lower performance.
Chapter 3

System Design

The following chapter describes our goals and assumptions, an overview of our system and design of the various components of this system.

3.1 Goals and Assumptions

The end goal is to be able to live migrate vFPGAs, but there is still a lot of groundwork that is needed before we can get to such a point. In this thesis, we simplify the problem by looking at how we can migrate a non-virtualized FPGA application. We further simplify the problem by making the following assumptions:

1. There is only one application running on the device;
2. The application and network are trusted, so no security mechanisms are considered.

Our main goal is to design a system for FPGA live migration that puts fewer limitations on how users design their hardware application compared to other related works. For example, the system in [30] requires hardware to be designed using OpenCL. We want to be able to migrate various types of hardware applications, ranging from those designed with RTL to those designed with HLS. Along with this, we want to enable these designs to have access to common FPGA features such as access to off-chip memory and
networking resources. Our secondary goal is to minimize downtime while performing a live migration.

3.2 System Overview

In our system, shown in Figure 3.1, the hardware application, which consists of the Memcached, TCP/IP, and AXI Interconnect, is placed inside a PR region on the FPGA. We expose an AXI and AXIS interface to the static region to give the application access to off-chip DDR3 memory and to an Ethernet network. For the purpose of demonstrating the migration of network connections, the application that is shown accesses the network through a TCP/IP stack. However, it is not limited to TCP and should work for any other protocol that runs on top of Ethernet.

We will first explain the migration process without pre-copying. When live migration is requested, the first step is to safely decouple the application from the static region. Decoupling happens in two stages. The first stage uses the AXI and AXIS decoupler blocks to decouple the application from the AXI and AXIS interfaces, respectively. Once these interfaces are decoupled, the second stage uses the Clock Gate block to stop the clock to the hardware application in the PR region.

Once the application is paused, the PR controller reads back the application’s configuration memory and stores it into off-chip memory. The configuration memory contains the current state of the application’s flip-flops, BRAM, LUT RAM and SRLs. The Bitstream Patcher unit merges this state back into the original bitstream to allow it to be used for programming the PR region in the destination FPGA.

At this point, we have all the state information that is required for restoring the application stored away in off-chip memory on the source FPGA; the next step is to transfer the state over to the destination FPGA. The Migration Engine transfers the patched bitstream and the application’s off-chip memory to the destination FPGA using
Figure 3.1: Block diagram of our system.
the 10G TCP/IP stack. The Migration Engine on the destination FPGA receives the data and stores it into off-chip memory.

The next stage is to restore the hardware application on the destination FPGA. The Migration Engine should have placed the application’s off-chip memory in the same location as it was on the source FPGA, so there is no need to move it to the correct memory location. The PR controller restores the hardware application by loading the patched bitstream into the PR region. The ARP Generator unit issues a gratuitous ARP to notify the switching network about the application’s new location so that the switch can redirect all incoming packets destined with the application’s MAC address to this new port.

The clock is then enabled, followed by coupling of AXI and AXIS interfaces to the destination FPGA’s static region. At this point, the application should begin running from its previous state before the move without being aware that it was migrated.

### 3.2.1 Minimizing Downtime

#### Memory Pre-copying

As we will show later in Section 4.3.2, transferring the application’s memory makes up the majority of the downtime during migration. We attempt to minimize downtime by using a technique from VM live migration called memory pre-copying [27]. Pre-copying allows for the VM’s memory to be copied to the destination server while the VM continues running on the source server. It attempts to copy the majority of the memory before pausing the VM, so that only a small amount is needed to be transferred when the VM is paused, thus reducing the overall downtime. Pre-copying is an iterative process where the memory is transferred over several rounds to the destination server. Since the VM is actively running while the memory is being transferred in each round, the memory on the destination server quickly goes stale as the VM makes new writes on the source server; the main idea of pre-copying is to only transfer these changes or deltas between
the two copies in the next round. The hypervisor keeps track of writes to memory and keeps a data structure in memory to indicate which pages were dirtied (written to), which it uses in the next pre-copying round to determine which pages to transfer over to the destination server. The VM is paused when the number of dirty pages from the previous round, also known as the dirtying rate, goes below a threshold or if there have been several rounds where no progress had been made in reducing the dirtying rate.

Pre-copying has the potential to minimize downtime, but it is highly dependent on the workload of the VM and usually does not work well with write-intensive workloads. There is a more deterministic and performant approach to minimizing downtime called post-copying \[37\] where the VM is first moved over to the destination server before any memory is transferred. Memory pages from the source server will be copied over when it is requested by the VM. Post-copying has been shown to perform better than pre-copying \[37\]. Google \[26\] has started to make use of both pre-copying and post-copying in their live migration process and the KVM hypervisor has plans to move to post-copying \[38\] in future releases.

Even though post-copying is a much better approach to pre-copying, we will start off with supporting pre-copying in our system first because it is much easier for us to implement in hardware. Our pre-copying system breaks the application’s memory into evenly spaced segments. We do not use virtual memory or have a memory management unit, but to be consistent with prior literature, we will refer to these segments as pages from this point forward.

We modify the original live migration flow by performing rounds of pre-copying before pausing the hardware application. The Write Snooper unit, shown in Figure 3.1, snoops the AXI AW channel between the application and the AXI interconnect to keep track of write requests to memory. The Dirty Bit Table is used to mark dirty pages during a specific round. The Write Snooper will mark the corresponding location in the table as dirty when it detects a write request. This table is double-buffered to prevent read
and write collisions. Also during the round, the Migration Engine walks the Dirty Bit Table, and retrieves and sends pages that are marked dirty to the destination FPGA. The Migration Engine also keeps track of the number of pages dirtied that round. This number is used to determine when to stop pre-copying and move to the next phase of migration. Once a round is completed, the tables are switched between the Write Snooper and Migration Engine.

3.3 Hardware Design

In the following section, we discuss the design of the various hardware components in our system.

3.3.1 AXI-Stream Decoupling

There are two AXIS interfaces crossing between the application’s PR region and the static region, one for receiving incoming data from the Ethernet IP core and one for transmitting data to the Ethernet IP core. Network packets are transferred as bursts of data on these interfaces. The TLAST signal marks the end of a packet in the data burst. Decoupling these interfaces during the middle of a packet transfer will lock up the transmit interface to the Ethernet IP and could also lock up the receive interface to the application. To safely decouple these interfaces, decoupling must only occur at a packet boundary, i.e., when TLAST is asserted. Xilinx provides an IP block called the Partial Reconfiguration Decoupler [39] for decoupling, but it is a simple decoupler with no concept of transactions or packet boundaries. As a result, we cannot use it for decoupling these two AXIS interfaces, and we will have to design our own decoupler.

To decouple an AXIS interface, the handshaking signals, TVALID and TREADY, must be driven to 0 to prevent the master and slave devices from registering a handshake. Our design uses two multiplexers, shown in Figure 3.2, to accomplish this. Normally, the
Figure 3.2: Multiplexing circuits for decoupling handshaking signals.

master registers a handshake when it asserts TVALID (\texttt{tvalid\_in}) and sees the slave asserting TREADY (\texttt{tready\_in}), and vice versa. However, to decouple these signals, we instead connect the master’s TREADY signal to \texttt{tready\_dc} and the slave’s TVALID signal to \texttt{tvalid\_dc}. We make use of a finite state machine (FSM) to ensure safe decoupling. When decoupling is requested and there is an ongoing transfer, the FSM will wait until the TLAST signal is asserted before decoupling the handshake signals. If there are no ongoing transactions, it will decouple them immediately.

### 3.3.2 AXI Decoupling

The application has access to off-chip memory by way of an AXI interface into the static region. Before moving on, we recommend going over Section 2.1.4 which covers some AXI concepts that we will be referring to in this section. Safely decoupling the AXI interface is more difficult than decoupling the AXIS interface in Section 3.3.1 because there are multiple channels that must be monitored, and multiple transactions may be in flight at any given moment whereas AXIS only has a single channel and a single transaction in flight at any moment. Improper decoupling, such as not waiting for all transactions...
to complete, will cause a deadlock on the bus. Xilinx provides an IP block called the Partial Reconfiguration AXI Shutdown Manager [10] for decoupling AXI interfaces safely. However, we have noticed that it buffers the last write transaction instead of preventing it from being accepted, and will complete it once the interface is coupled again. This is undesirable for designs such as ours where all write transactions are expected to have been completed and written back to off-chip memory upon decoupling. As a result, we cannot use Xilinx’s IP for decoupling the AXI interface in our design, and we must design our own custom decoupler.

Our AXI decoupler disconnects the read and write channels independently from each other. We use two counters, shown in Figure 3.3, one for the read channel and one for the write channel, to keep track of all outstanding transactions on the bus. We snoop the AR and AW channels and increment the read counter when we see a handshake on the AR channel and increment the write counter when we see a handshake on the AW channel. For reads, we snoop the R channel and wait for the RLAST signal to be asserted before decrementing the read transaction counter because the RLAST signal indicates the completion of a read transaction. For writes, we snoop the B channel and wait for an OKAY response before decrementing the write transaction counter because an OKAY response on the B channel indicates the completion of a write transaction.

The AXI specification [18] allows for write data to arrive on the W channel before the request on the AW channel. This complicates the logic needed for tracking write transactions and decoupling the write channels because we cannot solely rely on monitoring and controlling the AW channel to keep track of transactions and to prevent new ones from being accepted. Our design does not support this case, and we prevent it from happening by having a small FSM to realign the channels by preventing write data from being accepted before a write request on the AW channel has been accepted. Doing this is compliant with the AXI specification.

Similarly to decoupling an AXIS channel, an AXI channel is decoupled when its
handshaking signals, valid and ready, are driven to 0. We use the same multiplexing circuits used for decoupling AXIS, shown in Figure 3.2 for decoupling an AXI channel. The AXIS FSM, however, is not adequate for this design. Additional logic is required to support the counters and the multiple channels. There are two FSMs, one to manage the read related (AR and R) channels and the one to manage the write (AW, W, B) related channels. When decoupling is requested, both FSMs will decouple their respective request channels immediately (AR and AW) to prevent new transactions from being accepted. They will then wait for all transactions to complete by waiting until their respective transaction counters to reach zero at which point, the other channels will be decoupled.

3.3.3 PR Controller

The PR Controller is used for configuring and reading back the application’s PR region. It interfaces with the ICAP to access the FPGA’s configuration memory. The performance of the configuration and readback operations directly affects the downtime that is experienced during migration. The module’s performance is limited by the ICAP, which
Figure 3.4: A typical layout of a Xilinx FPGA bitstream [32].

Has a data width of 32 bits and can operate at a maximum frequency of 100MHz. That gives us a maximum throughput of 400 MB/s. The design of the PR Controller, at a high level, can be viewed as a high-performing DMA engine attached to the ICAP.

A bitstream write operation can be issued by specifying the base address of the bitstream in off-chip memory and bitstream size. The remaining details, such as fetching the bitstream from off-chip memory and communicating with the ICAP, are managed by the PR Controller.

A readback operation requires a few more steps than a bitstream write operation. The first step is to discover where the application’s configuration frames are located. Similar to what is done in [32], we analyze the application’s partial bitstream offline to extract metadata on the configuration frames. Figure 3.4 shows the layout of a typical bitstream file. A bitstream consists of several blocks of data, each containing configuration commands and followed by configuration data. The frame address register (FAR) parameter can be extracted from the configuration commands and it indicates the base frame address where the configuration data should be placed inside the FPGA’s configuration memory. The words parameter, which can also be found in the configuration commands, indicates the size of the configuration data in the block. The offset parameter indicates the file position where the configuration data begins in the bitstream file. We look for all configuration blocks in the bitstream file, and we extract and store the FAR, words, and offset parameters for each block into a metadata file.

Similar to configuration, readback requires us to issue commands to the ICAP to
request it to retrieve configuration data from a particular location in the FPGA’s configuration memory. These commands require us to specify which frame to begin reading from, and how many frames we want to read. The complete command sequence for readback is fully outlined by Xilinx in [17]. We use the FAR and words parameters in our metadata file to issue commands to read back the configuration data of the blocks that were specified in the initial bitstream file. Readback only returns the configuration data, and it does not contain any configuration commands. To create a working bitstream file from the configuration data, we replace the original bitstream’s configuration data with the readback data. We use the offset parameter that we extracted earlier to store the readback in the correct file position in the bitstream file. We use the metadata file to readback each block and store it back into the bitstream file.

### 3.3.4 Bitstream Patching

Programming the readback configuration data back into the FPGA should allow us to restore the previous configuration of the FPGA. However, we were not able to restore the contents of the BRAM and LUT RAM using the readback data directly to program the FPGA. The readback configuration data appears to be missing some control information that is used for loading initialization data into LUT RAM and BRAM [32]. Xilinx does not make the format of their configuration frames public, so we do not know what the commands are and where they are in the frame to issue the load request. Happe et al. worked around this problem by empirically determining where these configuration bits are located [32]. However, they target the Virtex 6 family of FPGAs, and we found that their formula did not work for the 7 Series. Furthermore, they did not describe how they derived this formula, so we are unable to derive a new one for the 7 Series.

However, we know that these commands are part of the original partial bitstream. We take advantage of this fact by developing a process to extract this information and patch them into the readback bitstream. Vivado can generate a mask file along with the
bitstream. The mask file indicates which bits of the bitstream should be ignored during verification of the readback data against the original configuration data. We can extract the configuration bits by inverting the mask file and using it to mask out the frames from the original partial bitstream. We generate a patched partial bitstream by applying the following bitwise operations to each word of all applicable frames:

\[
patched = (original \& \overline{mask}) \mid (readback \& mask)
\]

We are able to restore the state of flip-flops, BRAM, and LUT RAM with this patched partial bitstream. However, there may be some additional steps needed, since we had some issues trying to completely restore our application’s bitstream. We discuss this in Section 4.1.

The Bitstream Patcher unit is an accelerator for speeding up the patching operation. Initially, we used the MicroBlaze microcontroller for patching the bitstream but found it to be too slow, taking almost 10 seconds to patch a 13 MB file, and it became the biggest bottleneck in our system. We accelerate this operation because the bitstream patching directly affects downtime since patching can only be done when the application is paused. We use double buffering to hide off-chip memory access latency and process 128 words (512-bits) in parallel, allowing us to reduce the patching operation down to 10 ms.

### 3.3.5 Write Snooper

During each round in the pre-copying phase of migration, the Write Snooper is used to keep track of all writes that the application makes during the round. It does this by snooping the AXI AW channel between the application and the AXI Interconnect. Then it waits for a handshake on the channel, which indicates that a write request has been accepted. It then takes the address of the request and maps it to a page index. This page
index is dependent on the size of the page, a parameter that is configurable at runtime. The Write Snooper then uses this index to address the Dirty Bit Table to mark the page as dirty.

### 3.3.6 Dirty Bit Table

The Dirty Bit Table is used to keep track of dirty pages during the current and previous rounds of pre-copying. There are two tables, one for the Write Snooper to mark the dirty pages for the current round, and one for the Migration Engine to read from and send dirty pages from the previous round to the destination FPGA. We implement these two tables on-chip using true dual-ported (TDP) BRAMs to allow for single cycle access. There are a limited number of BRAM resources on the FPGA, which limits how big these tables can be. We size each table to 262,144 bits, which allows tracking of 1 GB of memory with a page size granularity of 4 KB.

We need to clear the table from the previous round to prepare it to be used by the Write Snooper in the next round. An efficient approach is to clear each entry from the table as the Migration Engine walks through it. This would mean having to write and read from the same address in the same cycle. The default settings for BRAMs when this happens is to return the write value on the read port, which is not useful for our use case because the Migration Engine will always be reading '0' from the table. Fortunately, we can override this setting to use a READ FIRST policy, which returns the current value stored in memory before overwriting it with the write value.

### 3.3.7 Migration Engine

The Migration Engine is used for sending and receiving state data between the destination and source FPGAs. There is a copy of the Migration Engine on each FPGA and they are connected over the network using the 10G TCP/IP network stack. The two Migration Engines work together in a client-server model where one acts as a client on the source
Figure 3.5: The Migration Engine header.

FPGA and the other as a server on the destination FPGA.

The client and server communicate using an 8-byte application-specific header. Figure 3.5 show the fields that make up the header. The client uses the header to initiate and end migration, and provide the server with information on the incoming data such as whether it is a bitstream or a memory page, its size, and its page offset, for use if it is a memory page. The payload is sent using multiple packets because it cannot fit in one packet; a TCP packet usually has a maximum payload size of 1460 bytes. The client only adds the header to the first packet. The server uses the header to acknowledge commands from the client.

The sequence diagram in Figure 3.6 shows the interaction between the client and the server during migration. Before migration starts, the server is idle and waits for the client to send it a start packet. Once the server receives the start packet, it sends an acknowledge packet to the client. The client enters the pre-copying mode where it begins sending dirty pages over to the server. The client uses a field in the header to let the server know the page offset so that the server can store it in the correct memory location. Once pre-copying is complete, the client waits for the bitstream readback and patching to complete before sending the patched bitstream over to the server. The client then sends a done packet to tell the server to restore the application on the destination FPGA. Once the application is resumed on the destination FPGA, the server sends an acknowledge packet to notify the client of this event.
Figure 3.6: Sequence diagram between the Migration Engines.
3.3.8 Clocking

There are three separate clock domains in our system. The majority of the system operates on the 156.25 MHz clock (core clock) generated from the 10G Ethernet IP. The 10G Ethernet IP core’s maximum datapath width is 64 bits, requiring a clock frequency of 156.25 MHz to achieve a bandwidth of 10G. We use the core clock to drive a BUFGCE clock buffer primitive to generate the clock (app clock) for the hardware application in the reconfigurable region. The BUFGCE primitive has a clock enable input that can be used to gate the application clock during the decoupling process. The AXI and AXIS interfaces are synchronized to the application clock domain. Since the application clock’s source is derived from the core clock, the phase shift between the two clocks is known, so we do not need synchronization circuits in the static region, allowing us to simplify the design of the decoupling blocks. The PR Controller operates in the second domain at 100MHz because the ICAP has a maximum operating frequency of 100MHz. The memory controller operates in the third clock domain at 166MHz. The logic for clock domain crossing (CDC) between these three different domains is automatically generated by the AXI Interconnect.

3.3.9 TCP/IP Network Stack

We make use of an open source hardware 10G TCP/IP networking stack [19, 20] in our system. It was designed using Vivado HLS, allowing us to easily make modifications to it to fit the needs of our application. We have two copies of this network stack in our system, one for the migration process and one for the application. The main modifications that we made to the one used in migration include:

1. Reduced the re-transmission timer to improve throughput between the source and destination FPGAs;

2. Reduced the number of supported sessions to save resources;
3. Added an AXI-Lite interface to allow the core to be configured at run time by a microcontroller.

3.3.10 ARP Generator

The ARP Generator is used to issue a gratuitous ARP on the destination FPGA after the application has been migrated to notify the network switch that the application’s MAC has moved to a different switch port. The MAC and IP parameters for the gratuitous ARP can be configured at runtime.

3.3.11 Controller

We designed software that runs on a MicroBlaze softcore processor to manage the migration process that we described in Section 3.2. Many of the different hardware blocks that were described previously are controlled by the MicroBlaze over an AXI-Lite Bus, shown in Figure 3.7. We map the control and status signals of each block to an addressable register space, and we add a 32-bit AXI-Lite interface to each block to allow this register space to be accessible by the MicroBlaze. The MicroBlaze makes it easier for us to prototype, debug and evaluate various parts of the system by allowing us to use software to modify the system on the fly without requiring us to rebuild the hardware.
Figure 3.7: The MicroBlaze controller.
Chapter 4

Evaluation

In this chapter we first describe what is functional in our system, followed by a description of our test setup. Afterwards, we present our results, and then we conclude the chapter with a discussion on what FPGA features are required for migrating FPGA applications.

4.1 Functionality

We have tested and confirmed that the majority of the system is working as designed. However, our system has a bug where we are unable to keep TCP sessions that were previously established open after restoring the patched bitstream on the destination FPGA. Initially, we designed and tested our system on a single FPGA and were able to keep TCP sessions open after restoring the bitstream. In our initial tests on the single FPGA, we:

1. Opened a TCP session;
2. Decoupled the interfaces and disabled the clock;
3. Read back and patched the bitstream;
4. Program the original partial bitstream into the PR region, which, in theory, should
clear out any residual configuration memory from the previous configuration;

5. Programmed the patched bitstream;

6. Coupled the interfaces and enabled the clock.

Afterwards, we sent some data over the opened TCP connection from the host to the FPGA, and we were able to receive a response. However, we have only recently tried restoring on a second FPGA and have just recently discovered the issue. Due to the lack of time, we are unable to debug this issue at the moment, but we suspect that we may be missing some state information with our bitstream patching process because it still functions correctly if we restored the bitstream on the source FPGA. This indicates that there might be some residual state information in the source FPGA’s configuration memory that was not cleared out even after applying the clearing bitstream. Further investigation is required to confirm this, and we leave that for future work.

We observe that the bug does not affect our ability to create new TCP sessions and make requests to the Memcached server on the destination FPGA. Furthermore, we have observed responses from the Memcached server that were queued inside the network stack while running on the source FPGA being sent out after the application resumed on the destination FPGA. These observations indicate that the patched bitstream is able to restore the application to some operable condition with much of its previous state intact. Also, we observe that the Memcached server is able to return cache hits for key-values that were stored while the server was still running on the source FPGA, which indicates that the memory was not corrupted during migration. We have further confirmed this by comparing the application’s memory contents on the source FPGA with those on the destination FPGA.

We have also encountered several bugs with the stock hardware Memcached source code in [28] that prevents the hardware Memcached server from operating at peak performance. These bugs limit us to a maximum of 100 simultaneous connections, and to a
certain key range; using keys outside of this range will lockup the server. We are aware that the hardware Memcached server has the potential to perform much better [42], but we will leave fixing these bugs to future work.

4.2 Test Setup

Our system requires an FPGA board with the following features:

- Xilinx 7 Series FPGA (Section 4.4 discusses why we need a 7 Series FPGA)
- Off-chip memory
- 2x SFP+ ports that can support 10G Ethernet

We use the Alpha Data ADM-PCIE-7V3 [43] board, which has a Xilinx Virtex-7 XC7VX690T FPGA, two channels of 8 GB of DDR3 memory, and two SFP+ ports that can support 10G Ethernet. Our test setup, shown in Figure 4.1, uses two 7V3 FPGA boards, one acting as the source FPGA and the other as the destination FPGA. Each board is installed into a PCIe slot of two separate servers. Both SFP+ ports on each board are connected to a 10G network switch. We configure the network switch to create two virtual local area networks (VLANs). VLANs allow us to connect specific switch ports together to create isolated local area networks. We use VLANs to isolate application traffic from management (migration) traffic to make it easier for us to debug network related issues, but it can potentially be used as a layer of security between the application and the management domains if the application is untrusted. The first VLAN is for the Migration Engines, and the other is for the application and its network clients. We connect an x86 server over a 10G link to another port on the application’s VLAN where it will act as a network client to the hardware application. In theory, our design can share one SFP+ port between the Migration Engine and the application, but we use two separate SFP+ ports for two main reasons. Firstly, it allows us to separate
the network traffic of the two hardware cores, making it easier to debug network related issues. Secondly, this separation should allow for the best performance because the two hardware cores do not need to share the bandwidth of a single 10G link. However, sharing a single SFP+ will reduce the total area overhead in the static region by removing the extra logic that’s needed to support a second 10G Ethernet link, but we would need to add bandwidth sharing hardware that can prevent the two cores from operating at peak performance.

For our migration system setup, the system is configured with a page size granularity of 1 MB and an application memory of 1 GB (1024 pages). The page size was picked arbitrarily, and further investigation is required to see if the page size has any performance impact on migration time.

4.3 Results

In this section, we breakdown the resource utilization of our system, describe our test methodology and present our results on migration time and the effectiveness of pre-
4.3.1 Resource Usage

Our system is placed on the FPGA as shown in Figure 4.2. The PR region is the highlighted outer box at the bottom left (the inner box is unimportant), and the remaining non-highlighted region represents the static region. We chose to place our PR region in that area because, through trial and error, we found it to be one of the better places for reducing routing congestion between the PR and static region. Routing congestion has been an ongoing problem for us and has occasionally made it very difficult or impossible
Table 4.1: Application utilization of the PR Region.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>73157</td>
<td>99200</td>
<td>73.8%</td>
</tr>
<tr>
<td>FF</td>
<td>99663</td>
<td>198400</td>
<td>50.2%</td>
</tr>
<tr>
<td>BRAM</td>
<td>263.5</td>
<td>350</td>
<td>75.3%</td>
</tr>
</tbody>
</table>

Table 4.2: Total utilization report.

<table>
<thead>
<tr>
<th>Resource</th>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Region</td>
<td>139176 (32.1%)</td>
<td>186187 (21.5%)</td>
<td>264.5 (18.0%)</td>
</tr>
<tr>
<td>Application</td>
<td>73157 (16.9%)</td>
<td>99663 (11.5%)</td>
<td>263.5 (17.9%)</td>
</tr>
<tr>
<td>Total</td>
<td>212333 (49.0%)</td>
<td>285850 (33.0%)</td>
<td>528 (35.9%)</td>
</tr>
</tbody>
</table>

to route our design. The region we chose consists of 32548 configuration frames (13 MB). Table 4.1 shows the resources available in the PR region, and how many of these resources are used by the application.

The total resource usage by all the components in the system including the application is shown in Table 4.2. We use less than half of the resources on the FPGA, and the application takes up less than 20% of the resources, potentially allowing for larger applications or more than one application to reside on the FPGA.

The static region consists of the migration infrastructure and other supporting hardware, such as the AXI interconnects, memory controllers and Ethernet subsystems. Table 4.2 shows the total utilization of the static region. We breakdown the utilization of the components in our migration infrastructure in Table 4.3. The migration infrastructure makes up around 40% of the total LUT and FF usage of the static region, and it is the main consumer of BRAM in the static region. However, in terms of total resources of the whole FPGA, the migration infrastructure takes up a small part of the total FPGA resources, using less than 16% of FF, LUT and BRAM resources. We do not show DSP usage because we do not use any of this resource. The Clock Gate unit only instantiates a clock buffer primitive, and does not need any other resources, which is why the table shows no resource usage for the Clock Gate. The largest component is the TCP/IP net-
Table 4.3: Migration infrastructure utilization breakdown.

<table>
<thead>
<tr>
<th>Resource</th>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW Core</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10G TCP/IP Stack</td>
<td>34973 (8.1%)</td>
<td>44966 (5.2%)</td>
<td>101 (6.9%)</td>
</tr>
<tr>
<td>ARP Generator</td>
<td>496 (0.1%)</td>
<td>599 (0.1%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>AXI Decoupler</td>
<td>108 (0.0%)</td>
<td>41 (0.0%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>AXIS Decoupler</td>
<td>15 (0.0%)</td>
<td>16 (0.0%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Bitstream Patcher</td>
<td>8969 (2.1%)</td>
<td>20902 (2.4%)</td>
<td>73.5 (5.0%)</td>
</tr>
<tr>
<td>Clock gate</td>
<td>0 (0.0%)</td>
<td>0 (0.0%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Dirty Bit Table</td>
<td>44 (0.0%)</td>
<td>12 (0.0%)</td>
<td>16 (1.1%)</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>1351 (0.3%)</td>
<td>1138 (0.1%)</td>
<td>8 (0.5%)</td>
</tr>
<tr>
<td>Migration Engine</td>
<td>6466 (1.5%)</td>
<td>7975 (0.9%)</td>
<td>16 (1.1%)</td>
</tr>
<tr>
<td>PR Controller</td>
<td>2813 (0.6%)</td>
<td>3788 (0.4%)</td>
<td>10 (0.7%)</td>
</tr>
<tr>
<td>Write Snooper</td>
<td>135 (0.0%)</td>
<td>158 (0.0%)</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>55370 (12.8%)</td>
<td>79595 (9.2%)</td>
<td>224.5 (15.3%)</td>
</tr>
<tr>
<td><strong>% of Static Region</strong></td>
<td>39.8%</td>
<td>42.8%</td>
<td>84.9%</td>
</tr>
</tbody>
</table>

Table 4.4: Memtier benchmark configuration.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Threads</strong></td>
<td>10</td>
</tr>
<tr>
<td><strong>Connections per thread</strong></td>
<td>10</td>
</tr>
<tr>
<td><strong>Requests per client</strong></td>
<td>100000</td>
</tr>
<tr>
<td><strong>Total number of requests</strong></td>
<td>1000000</td>
</tr>
</tbody>
</table>

work stack, which we cannot do much to reduce in size other than reduce the number of sessions that it can support to lower BRAM usage. However, it has the capability to be shared among other components in the static region to gain access to the network. The Bitstream Patcher is the second largest block. We can reduce the area at the expense of lowering performance (increasing the patching time) by reducing the number of words that we can process per cycle. The migration infrastructure’s area utilization should remain the same when we scale up to larger FPGAs because no component in the infrastructure is dependent on the size of the FPGA. However, the area of the pre-copying hardware, i.e., the Write Snooper, Dirty Bit Table, and Migration Engine, will go up if the application’s memory is increased because we will need more BRAM and logic to track the additional pages. However, increasing the page size may help to offset this cost by reducing the number of pages that are required to be tracked.
4.3.2 Migration Performance

In this section we measure the performance of our migration system. The setup for the application region is shown in Figure 3.1. We use the hardware Memcached server that we described in Section 2.4 as our test application. The Memcached server is connected to the network using a copy of the hardware 10G TCP/IP network stack. We use TCP to ensure that any packets that are dropped while the application is paused will be retransmitted and sent to the destination FPGA once the application resumes after migration. We use memtier benchmark [44], a Memcached benchmarking tool, to make requests to the Memcached server. Table 4.4 shows our memtier benchmark configuration. We are limited to 100 simultaneous connections because of a bug in the Memcached server, that we have described in Section 4.1.

Performance Impact on Application Throughput

A metric that we want to measure is the performance impact on the application while performing a migration. Latency and throughput are important factors when characterizing a Memcached server, and both should be impacted negatively when performing a migration. However, we want to characterize our migration system, not Memcached, so we only focus on the impact of one of these metrics, throughput. The throughput of a Memcached server is measured by how many operations it can perform per second (Ops/sec). We first obtain a baseline, shown in Figure 4.3 by measuring the performance of the server before migration by running memtier benchmark. We measure throughput by configuring memtier benchmark to perform a number of operations on the Memcached server, for all our measurement it is 10,000,000 operations. Memtier benchmark keeps track of how long it takes to perform all these operations, and it uses this time to calculate the overall throughput after all operations are complete.

Firstly, want to see the impact on throughput during the worst case migration scenario that results in the longest downtime. In this scenario, which we will call stop-and-copy, no
pre-copying is performed before pausing the application and the entire 1 GB of application memory is transferred over to the destination FPGA while the application is paused. We cannot set this scenario up using our system because, as we discussed in Section 4.1, we are unable to keep TCP connections open after restoring on the destination FPGA. However, we can closely simulate this event by pausing the Memcached server on the source FPGA for the amount of time we expect the migration to take and then resume it. After starting the memtier_benchmark tool, we pause the Memcached server by decoupling and clock gating it. We wait 1000 ms, which is an accurate representation of downtime for this scenario as we will show later on, to simulate the migration process, and then we couple and enable the clock again. The result is shown in the middle bar in Figure 4.3. It shows that there is around a 9.5% decrease in throughput during a stop-and-copy migration.

Secondly, we want to see if pre-copying has any negative effects on the application’s performance. It has been shown in VM migration [27] that pre-copying can reduce a VM’s performance by reducing the available memory and network bandwidth to the VM.
This impact is mitigated by limiting the bandwidth allocated to the pre-copying process. Analogously, we want to see if the memory bandwidth used by pre-copying is affecting the performance of our application and determine if we need to add bandwidth limiting mechanisms into our design. We expect this to be the case since Memcached is a memory intensive application. The worst case where pre-copying consumes the most memory bandwidth is when the application dirties the entire page table each round, forcing the Migration Engine to transfer the entire page table each round. We set up this scenario by modifying the Migration Engine to ignore the dirty table, and send every single page out during each round of pre-copying. The result is shown by the rightmost bar in Figure 4.3. It shows a 0.7% decrease compared to the baseline which seems to indicate some performance degradation. However, we believe this can be attributed to factors of uncertainty in our measurement, such as the host OS running the benchmark and how busy the network switch is at the moment. We do not have mechanisms in place to measure the memory bandwidth usage, but we surmise that the aggregate bandwidth used by the application and the rest of the components in the static region is much lower than the available 10.6 GB/s of bandwidth that the off-chip memory is capable of providing. We believe this because our Memcached server cannot operate at its peak performance and the two 10G TCP/IP network stacks only have a maximum aggregate bandwidth of 2.5 GB/s (2x10Gbps/8). However, we do expect the performance to be impacted with a fully functioning Memcached server because it should make more significant use of the memory bandwidth.

Pre-copying Effectiveness

In this section, we evaluate how effective pre-copying is for reducing application downtime during migration. Downtime is defined as the time from pausing the application on the source FPGA to when it is resumed on the destination FPGA. As we have shown in the previous section, downtime can reduce our application’s performance, so we want to
minimize downtime as much as possible during migration. The operations that happen during downtime are:

1. Readback of the bitstream;

2. Patching the bitstream;

3. Transferring the remaining dirty pages of memory to the destination FPGA;

4. Transferring the patched bitstream to the destination FPGA;

5. Restoring the bitstream on the destination FPGA.

We first measure and breakdown the downtime during a stop-and-copy scenario where there is no pre-copying to see how much memory transfer adds to our downtime. The result is shown by the top bar in Figure 4.4. The downtime, which in this scenario is also the total migration time, is a little over 1000 ms. The majority of that time is dominated by transferring the application’s 1 GB of memory. The readback operation takes more time than the restore operation because we need to issue both write and read operations to the ICAP during readback, which incurs some additional overhead, while we only need to issue write operations to the ICAP when doing a restore. We can reduce the transfer time slightly by overlapping the memory transfer with the bitstream readback and patching operations.

As we saw, almost 90% of the downtime is attributed to memory transfers when performing a stop-and-copy migration. We will attempt to lower this by using pre-copying to try to reduce the amount of memory we require to transfer while the application is down. The breakdown of downtime when using pre-copying is shown in the bottom bar in Figure 4.4. The downtime has now reduced from 1000 ms to approximately 150 ms, an 85% reduction. It shows that memory transfer is no longer the biggest contributor to downtime, surprisingly, it is now the smallest part of the downtime. The downtime is now dominated by the patching and restoring process. We can further reduce the downtime by
optimizing our PR Controller since it is only operating at half of the 400 MB/s bandwidth that the ICAP can support. Though these numbers show significant improvements to downtime when using pre-copying, we expect the improvements to be much less for other workloads because our Memcached server is not operating at peak performance, resulting in a low dirtying rate, which is a scenario where pre-copying performs the best [27, 37]. Our measurements show that after transferring the initial 1024 pages (1 GB), the dirtying rate dropped and consistently remained at 9 pages per round.

There is no other work, to the best of our knowledge, that is similar to ours that we can use to compare our downtime against. The closest comparison that we can make against is with the downtime that is experienced with VM live migration. We are aware that this is not a good comparison because these systems are vastly different from each other in terms of architecture (software versus hardware), network infrastructure, and test setup. The work by Song et al. in [45] demonstrates a technique to parallelize the VM live migration process. They achieved a downtime of 115 ms while migrating a
Memcached server with 16 GB of memory running on a KVM hypervisor.

We attempt to further investigate the effectiveness of pre-copying as a mechanism for reducing downtime for FPGA migration. We know from previous work [27, 37] that pre-copying may not be very effective for write-intensive workloads because it prevents the dirtying rate from lowering. We try to see if this is also the case for FPGA migration by using a synthetic application that has two modes. A typical memory access pattern by the FPGA application usually involves burst reads and writes to memory to offset the off-chip memory latency. The first mode, which we will call sequential, tries to simulate this by performing sequential burst writes to the entire memory space. The second mode, which we will call destructive, attempts to dirty pages as fast possible by continuously making only single writes to each page. We replace the Memcached application with this synthetic application in our system and perform iterative rounds of pre-copying from the source to the destination FPGA. The measured dirtying rate at each round of pre-copying is shown in Figure 4.5. In round 0 we transfer all the pages and begin tracking dirty pages. From round 1 and onwards we only send the dirty pages. For sequential mode, we can see that the dirtying rate converges quickly to 2 pages per round after only 3 rounds. For destructive mode, the dirtying rate converges to around 700-800 pages per round. Our pre-copying algorithm will terminate when it sees 3 rounds of no change in the dirtying rate or when it sees a rise in the dirtying rate. In the destructive mode scenario, after seeing the dirtying rate jump in round 7, it will stop the application in round 8 and do the final transfer. This adds an additional 535 ms to the downtime for transferring the remaining pages while the application is paused. Based on our preliminary results, pre-copying appears to more effective for reducing downtime for migrating FPGA applications that mostly perform burst writes to memory shown by the sequential benchmark. This could be because these burst writes fall into the same or adjacent pages resulting in a low dirtying rate. However, we need to perform more thorough testing to confirm this, and we leave that to future work.
4.4 FPGA Features for Live Migration

As we mentioned in Chapter 1, FPGAs are rapidly being deployed in the data center, but current FPGAs were not designed with data centers in mind. As a result, they lack features that would make them easier to manage and deploy in such an environment. In this section, we provide suggestions and ideas for features that can be implemented in future FPGAs to help make it easier to support FPGA migration in the data center.

We show in Section 4.3 that with pre-copying the majority of the downtime when migrating is from the bitstream readback and restore operations. We also show that we can achieve relatively fast readback and restore times with our PR Controller, which only uses half of the ICAP’s bandwidth for configuration. Based on these findings, we believe that bitstream readback and restore can be a promising hardware preemption mechanism for FPGA migration because it is fast and flexible by allowing for different types of hardware designs to be preempted. We suggest that a similar feature be added.
to future FPGAs designed specifically with hardware preemption in mind. We would like to see a more turnkey solution for hardware preemption where the FPGA configuration architecture has built-in support for configuration readback and restore, and the tools have built-in support, such as IP blocks and software drivers, for designers to make use of this feature because the current readback functionality in Xilinx FPGAs is not designed for hardware preemption and is meant for verification purposes. This is clearly evident because the readback technique that we use to store state in a Virtex 7 device is incompatible with the newer UltraScale generation of FPGAs; Xilinx has updated their configuration architecture to not use the initialization memory cells for reading back flip-flops [46]. The UltraScale architecture now has dedicated configuration memory cells for readback of registers to avoid overwriting initialization memory that may be needed by the user’s hardware design. We restore state by issuing a command to initialize registers from their initialization memory cells after programming the bitstream. Because of this, we require the register states to be stored into initialization memory when performing a readback, but this is not the case in the UltraScale. Some features that this new readback and restore mechanism should have are:

1. A new or updated configuration interface similar to the ICAP to allow for faster access to the FPGA’s configuration memory to minimize downtime;

2. Readback data should be in a state that is ready to be used to configure the FPGA again without having to patch it;

3. Support for readback and restore of DSP and BRAM registers, which will allow for more designs, mainly those that use DSPs, to be migrated.

Vendors may also need to update or create new IP Blocks to better support migration. We needed to build our own custom decouplers and ICAP controller because the Xilinx equivalent IP blocks did not have features, such as packet-based decoupling for AXIS, that we required to safely prepare the application for migration. Also, we suggest that
Interconnect IP blocks should have Quality of Service (QoS) capabilities to make it easier to manage bandwidth allocation.
Chapter 5

Conclusion

In this thesis, we present a system for live migrating network connected FPGA applications. With this system, we show that it is possible to live migrate FPGA applications from one physical FPGA to another physical FPGA using the capabilities, such as readback, that are available on current FPGAs. However, we have also shown that these capabilities were not designed with migration in mind and that they are difficult to use for migration, requiring a lot of reverse engineering and guesswork. We have shown that by using readback and restore as a state saving mechanism, we can build a flexible migration system that can migrate different kinds of hardware designs, even those that are network connected. By using only half of the ICAP’s bandwidth, we have demonstrated that readback and restore can be relatively fast operations during live migration, allowing for a downtime of less than 200 ms for a moderate size bitstream (13 MB). Because of these two reasons, we believe that readback is a viable state saving mechanism for live migrating FPGA applications and we believe that future FPGAs should have a readback feature that is designed specifically with hardware preemption in mind to allow live migration to more streamlined and more performant on FPGAs.

Our preliminary results have shown that there may be some benefits to using precopying to reduce the downtime for certain FPGA applications, however, further investi-
Chapter 5. Conclusion

5.1 Future Work

There are several bugs, described in Section 4.1 that prevented us from fully demonstrating and evaluating the full capabilities of our system. We plan to address and fix these bugs in the near future.

In Section 4.3, we mentioned that we did not have the capabilities to limit or measure the memory bandwidth usage by the application and the migration infrastructure. We want to add a bandwidth monitor to allow us to have better insight into where the memory bandwidth is begin allocated. Following that, we may want to add a Quality of Service mechanism such as those described by Rozhko in [9] to have more control over the bandwidth allocation since it has been shown that in VM migration, the migration process can affect the VM’s performance by stealing memory bandwidth [27].

We did not have enough data to make a conclusion on the effectiveness of pre-copying on reducing downtime while migrating FPGA applications. We may want to further investigate this by measuring other types of applications. We may also try to investigate and implement post-copying as a mechanism for reducing downtime for FPGA migration since it has been shown to outperform pre-copying [37]. Unlike pre-copying, post-copying will require us to readback and restore the application’s state on the destination server first before transferring the memory. Afterwards, the memory will be transferred over to the destination FPGA. If the application requests a memory page that has yet to be transferred over to the destination, it will generate a network fault, which will require immediate retrieval of that page from the source FPGA to minimize the amount of time
the application is stalled. To accomplish this, we will need an additional hardware block, similar to the Dirty Bit Table, to keep track of the pages that have been sent over to the destination FPGA to allow us to check if we need to generate a network fault to retrieve the page from the source FPGA. We will also need to modify the Migration Engine. Firstly, we will need to reverse the roles of the Migration Engines after transferring the bitstream, where the Migration Engine on the source FPGA becomes the server, and the one on the destination becomes the client to allow the destination to request pages from the source FPGA. Also, we will need to add logic to allow the Migration Engine to issue page requests in client mode and to retrieve page requests in server mode.

Our live migration system could be used in a different way to minimize the impact of an unexpected FPGA failure. Instead of performing a full migration, we can instead use it to make periodic checkpoints of the application’s memory and hardware state. Without checkpointing, an FPGA failure will result in a complete loss of the application’s data on the FPGA. However, with periodic checkpointing, failure is not as catastrophic since we can potentially recover most of the data by rolling back and restoring a previous checkpoint. These checkpoints can be stored to a server and can be retrieved when needed.
Bibliography


