HIGH-LEVEL SYNTHESIS OF DATACENTER SERVICES

by

Justin Tai

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

© Copyright 2017 by Justin Tai
Abstract

High-Level Synthesis of Datacenter Services

Justin Tai
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2017

Field programmable gate arrays have become of great interest for implementing data-center applications due to high performance gains over traditional compute hardware at a fraction of the energy consumption. In parallel, more hardware designers are adopting high-level synthesis for its rapid development process. It stands to reason that high-level synthesis tools will be widely used to build datacenter applications in the near future.

In this work, we propose that entire datacenter services be implemented using only field-programmable gate arrays and present a case for this. We propose and provide prototype support for an enhancement to a production high-level synthesis tool, Vivado HLS, which allows users easily to build distributed systems strictly in hardware. Finally, we demonstrate operation of our feature by implementing a simplified version of web search using our enhancement.
Acknowledgements

I would like to thank my supervisor, Professor Paul Chow, for the opportunity to do this master degree under his tutelage. With his guidance, I was able to brainstorm and develop my own ideas, an experience that I strongly value. It was also interesting to see how research is conducted and the advancements in technology that are being made today.

I thank my parents for their support, both financially and emotionally. Without them, this process would have been much more difficult.

I had the pleasure of meeting many friends along the way: Daniel Ly-Ma, Eric Fukuda, Roberto DiCecco, Daniel Rozhko, Sanket Pandit, Ruediger Willenberg, Jin Hee Kim, Julie Hsiao, Joy Chen, Naif Tarafdar, Jasmina Vasiljevic, Fernando Martin del Campo, Xander Chin, Ehsan Ghasemi, Keiming Kwong, and Vincent Mirian. I hope that we will stay acquainted for many years to come.

During my tenure as a graduate student, I was privileged to be able to work as a teaching assistant at the University of Toronto. I thoroughly enjoyed interacting with all the students that I encountered, and thank them for making my time at the university livelier.

Finally, I would like to thank Xilinx Inc. and Rogers Communications for funding this research.
Contents

1 Introduction ................................................. 1
   1.1 Motivation ........................................... 1
   1.2 Contributions ....................................... 2
   1.3 Thesis Overview ..................................... 2

2 Background .................................................. 4
   2.1 The Datacenter Environment ......................... 4
   2.2 Datacenter Applications .............................. 6
   2.3 Distributed Systems ................................. 8
   2.4 High-Level Synthesis ............................... 11

3 Related Work ............................................... 13
   3.1 OpenMP ............................................... 13
   3.2 Distributed Memory Compilers ....................... 15
   3.3 FPGAs in Datacenters ................................ 16

4 High-Level Synthesis of Datacenter Services .......... 17
   4.1 The Case for Datacenter Services on FPGAs .......... 17
   4.2 Programming Model ................................... 19
      4.2.1 Pragma Syntax ................................ 22
      4.2.2 Generated IP Cores ............................. 23
      4.2.3 User Guarantees ................................ 24
   4.3 Using the IP Cores ................................... 25
      4.3.1 Worker Replication ............................ 26
      4.3.2 Worker Sharing ................................ 28
      4.3.3 Deployment in the Datacenter .................. 30
      4.3.4 Building Large Scale Distributed Systems ...... 32
   4.4 Examples .............................................. 32
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Summary of C constructs that are supported by HDS and the information that Pragma Parser collects in regards to each of them.</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>Summary of supported modification requests.</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Categorization of supported and unsupported data types. Note that multidimensional arrays are supported, but only of the supported data types.</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>The number of instances of each IP block in our design.</td>
<td>58</td>
</tr>
<tr>
<td>5.5</td>
<td>Resource utilization of Add N with and without injected communication and aggregation logic. Unused resources have been omitted.</td>
<td>60</td>
</tr>
<tr>
<td>5.6</td>
<td>The number of instances of each IP block in our design.</td>
<td>65</td>
</tr>
<tr>
<td>5.7</td>
<td>Percent error between the average number of search results at each data-point and the expected number of results.</td>
<td>67</td>
</tr>
<tr>
<td>5.8</td>
<td>Percentage difference between ideal and experimental query execution latencies.</td>
<td>68</td>
</tr>
<tr>
<td>5.9</td>
<td>Resource utilization of web search with and without injected communication and aggregation logic.</td>
<td>69</td>
</tr>
</tbody>
</table>
List of Figures

2.1 One of Google’s datacenters as seen from outside. Google’s datacenters are large warehouse-like structures. Note the scale of the construction vehicles, center, relative to the size of the building. Taken from [1].

2.2 Inside one of Google’s datacenters. Google’s datacenters are populated with row upon row of server racks. In the image, each row is comprised of many server racks, and each rack holds many server blades. Taken from [2].

2.3 A pie graph decomposing the power draw of Google’s datacenters. CPU power consumption dominates, accounting for 42% of the power draw. Taken from [3].

2.4 Diagrams showing a decentralized (left) and a centralized (right) application.

2.5 Diagram depicting the master worker model. Masters commit tasks to the network without any knowledge of how many workers there are.

2.6 Implementation of applying the $f(x) = 2x + 1$ to all elements of an array

3.1 A trivial example demonstrating the use of the parallel pragma in OpenMP. “Hello World!” will be printed one or more times, once for each spawned thread.

3.2 Example usage of the for directive. “Hello World!” will be printed exactly ten times.

4.1 Simple implementation of adding one to every element to an input array

4.2 Block diagram showing the master-worker implementation of the add one application in Figure 4.1.

4.3 Example callgraph demonstrating vertical scaling. fooA() is the top level function. fooB() and fooC() are distributed functions. Both fooA() and fooB() will generate request messages when “invoking” fooC().

vii
4.4 The syntax of the DISTRIBUTE pragma. When a function is labelled with the DISTRIBUTE pragma, it is implemented as a separate IP core. The user may optionally specify the partition size and elements variable if they wish to partition execution of the function.

4.5 Example usage of the DISTRIBUTE pragma, applied to the add one example shown in Figure 4.1.

4.6 A generated distributed block. The DISTRIBUTE pragma has added the input/output AXI stream ports, the M_AXI ports, and the ID port.

4.7 Distributed implementation of times two add one.

4.8 Block diagram of small scale implementation of $f(x) = 2x + 1$. top_level is connected to one instance of add_one and times_two using a multiplexer and a demultiplexer.

4.9 Block diagram with top_level pushing work to multiple worker instances.

4.10 Diagram showing the flow of packets with one instance of top_level and multiple instances of add_one and times_two. top_level outputs a stream of requests that are routed to the scheduler. The scheduler load-balances incoming requests to the add_one and times_two workers.

4.11 Diagram showing multiple top_level instances committing work to multiple add_one and times_two instances. The scheduler accepts requests from all instances of top_level. It load-balances the requests to instances of add_one and times_two. The workers directly communicate their results back to the original requesting top_level instance.

4.12 Block diagram showing multiple top_level instances pushing work to multiple times_two and add_n instances.

4.13 Diagram showing different kinds of masters, top_level() and top_level_new(), sharing the same worker nodes, addOne() and times_two().

4.14 Block diagram of one possible configuration of our distributed blocks. Each FPGA may hold a different mix of our distributed blocks. The distributed blocks may interact with each other over any communication medium, at any scale, including the Internet.

4.15 A diagram depicting the overall basic algorithm of web search, irrespective of how it is distributed.

4.16 A diagram depicting the architecture of the Google search engine.

4.17 Code showing the basic structure of Google Web Search implemented using the DISTRIBUTE pragma.
4.18 Here, a block diagram of FlyWheel is shown. The fetch service is separate from FlyWheel and used by other services. ............................. 38
4.19 Skeleton pseudo-code for FlyWheel ............................................. 39

5.1 Template for aggregation functions, adjusted for readability. .......... 43
5.2 Template for worker functions, adjusted for readability. ............... 44
5.3 Figure depicting communication protocol between masters and workers. 44
5.4 Declaration of the packet_t structure and call signatures for the send() and recv() functions for transmitting a character (signed 8-bit value). Adjustments were made for readability. ........................................... 46
5.5 The parts of our source-to-source transformation tool and how they interact with each other. ......................................................... 47
5.6 Major steps of the transformation algorithm. ............................... 49
5.7 Example of a data dependency graph. ........................................... 50
5.8 Code snippet implementing summation of all elements of an array. ... 51
5.9 Shown is a for-loop definition that HDS has adjusted in an attempt to maintain functional equivalency. ........................................... 52
5.10 Skeleton code demonstrating DISTRIBUTE nesting. ..................... 53
5.11 Block diagram showing hierarchical master-worker relationships. The acts of distribution and aggregation have themselves been distributed. ........ 54
5.12 Block diagram showing the hierarchical structure of Dremel, Google’s large scale SQL processing system. Taken from [4]. .......................... 54
5.13 Source code for our add N microbenchmark. Pragma already part of the Vivado HLS 2016.2 [5] specification have been omitted for brevity. Our contribution is only the DISTRIBUTE pragma. ......................... 56
5.14 Block diagram of our Add N microbenchmark. The ZYNQ processing system exists only for configuring the system and timing execution. .... 58
5.15 Graph showing the execution times of the Add N microbenchmark as the number of workers are increased. As expected, there are diminishing returns as the number of workers increases. .......................... 59
5.16 Diagram showing correct aggregation of an array that has been linearly partitioned. Grey represents valid data, whereas white represents unused memory or nonsensical data. ................................. 63
5.17 Diagram showing correct aggregation of an array where each partition has invalid data. Grey represents valid data, and white represents unused memory or nonsensical data. ................................. 63
5.18 Architectural view of our experimental web search. The core distributed system has been coloured red while paths associated with off-chip DDR have been coloured blue. ................................................................. 64
5.19 Block diagram of our web search implementation with 4 partitions of the reverse index and URLs table. The paths corresponding to DDR connections are shown in blue. ................................................................. 65
5.20 layout of DDR memory. ................................................................. 66
5.21 Execution time of web search as the number of partitions is increased. ................................................................. 68
6.1 Tree of possible research directions. ................................................................. 72
Chapter 1

Introduction

1.1 Motivation

Field Programmable Gate Arrays (FPGAs) have garnered significant interest from datacenter stakeholders recently. In 2014, Microsoft announced Catapult [6], the first commercial deployment of FPGAs in datacenters. The year after, Intel acquired Altera, a $16.7 billion venture, in an effort to defend its stake in the datacenter market [7]. The reason: FPGAs, if leveraged properly, have been shown to deliver multiple times more performance than traditional compute hardware, namely Central Processing Units (CPUs), at under a tenth the power consumption [6]. It is the latter, low power consumption, that make FPGAs compelling. With the end of Dennard scaling, power consumption is the limiting factor for datacenter growth today. Energy is an increasingly expensive commodity, and cooling solutions are at their limits [8].

Simultaneously, High-Level Synthesis (HLS) tools have also risen in popularity. HLS tools are those that translate programming languages such as C, C++, and Fortran into Hardware Description Languages (HDLs) like Verilog, VHDL, and SystemC. The ability to express hardware algorithmically (i.e., what algorithm the hardware should implement), as opposed to explicitly specifying what hardware should be built, greatly expedites the development process. This has two major advantages. First, building applications in hardware becomes economically viable since development time, and therefore development cost, is reduced. Writing Register Transfer Logic (RTL) is analogous to writing assembly code – onerous and prone to error. Second, better designs can often be produced when using HLS tools because it is easier to explore alternatives. When writing HDL, one may be forced to settle on a sub-optimal design as a time saving measure.
Given that FPGAs are likely to be incorporated into datacenters and that HLS is becoming widely adopted, an interesting extrapolation is that HLS tools may be used to build datacenter applications soon. A natural question to ask is: “How can HLS be tailored to facilitate implementation of datacenter applications on FPGAs?” In this work, we propose and explore one such enhancement to HLS.

1.2 Contributions

The goal of our work is to automate the process of building distributed applications in hardware, with the intention that entire datacenter applications will be implemented on FPGAs using HLS. We propose an enhancement to the HLS programming model that allows users to describe distributed systems. Developers annotate C/C++ functions with pragmas to define the constituents of the distributed system and the relationships between them. We provide proof-of-concept support for our proposed HLS enhancement in the form of a source-to-source transformation tool. We evaluate our work by implementing a microbenchmark, as well as a simplified version of web search using our enhancement. In summary, our contributions are two-fold:

- An enhancement to the Vivado HLS programming model enabling users to describe distributed systems;
- Prototype support for said enhancement as a source-to-source transformation tool based on Clang/LLVM.

To our knowledge, our vision of datacenter services fully implemented on FPGAs, and our enhancement to HLS are novel.

1.3 Thesis Overview

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of the datacenter environment, the applications that run in them, the master-worker model that is common to datacenter applications, and high-level synthesis tools and their programming models. In Chapter 3 we provide a sense of the landscape of work that has been done in this area. Most notably, we differentiate our work from OpenMP, which bears some resemblance to our research. In Chapter 4, we introduce our contribution, the addition of the DISTRIBUTE pragma to Vivado HLS. Here we motivate its need, give its syntax, describe the hardware it generates, and provide simplified real-world
applications implemented using our model. Having discussed our programming model and its uses, we present our prototype support for it in Chapter 5, intended as proof-of-concept. We give details on the transformation algorithm we use, and the architecture of our source-to-source transformation tool. We demonstrate correct operation of our work by implementing a microbenchmark and a simplified version of web search using our enhancement. We conclude by leaving the reader with outstanding tasks and interesting directions for future work in Chapter 6.
Chapter 2

Background

In this chapter, we give background information crucial to the reader’s understanding of our work. We begin by describing datacenters, how they operate, their design concerns, and the applications that run in them. We next describe the master-worker distribution model, which is the focus of our work. Many datacenter services follow this distribution model because its design elegantly addresses many of the concerns that arise when building datacenter applications. Finally, we provide a brief overview of HLS tools and more specifically, its programming model.

2.1 The Datacenter Environment

Our work aims to ease the process of building applications that would run in datacenters using FPGAs. We set the scene for this thesis with a brief discussion on datacenters.

Datacenters are clusters of collocated computers, often housed in warehouse-like buildings. Today, datacenters have grown to massive scales due to a paradigm shift from client to server side computing, colloquially known as cloud computing. This shift has put unprecedented levels of pressure onto datacenters. A mid-sized Google datacenter today houses some 10,000 compute nodes [9]. Each node in a datacenter is a full computer, housing its own CPU, RAM, persistent storage, high-speed network link, and cooling solution in its enclosure [8]. Figures 2.1 and 2.2 show how Google’s datacenters look from the outside and inside, respectively. The sheer scale of modern datacenters has three major implications.

First, it is critical to minimize the dollar spent to unit performance ratio of each node in the system [8]. For the largest datacenter operators, reducing the cost of every node
Second, applications running in the datacenter must be fault tolerant. Running such large quantities of servers, datacenters today see a server failure once every few hours [3]. Failure rates are exacerbated by the fact that cheaper, less durable parts are used in datacenters as a cost saving measure.

Third, nodes in systems may vary on the dimensions of CPU, RAM, and disk space [9]. This is a result of older parts being gradually replaced with newer, more capable com-
Figure 2.2: Inside one of Google’s datacenters. Google’s datacenters are populated with row upon row of server racks. In the image, each row is comprised of many server racks, and each rack holds many server blades. Taken from [2].

...ponents as they fail or become obsolete. Clever scheduling of work onto these servers is necessary to maximize utilization while minimizing latency.

In almost every discussion regarding datacenters, the issue of power consumption seems to come up. Today, power consumption and its byproducts are the factors stinting the growth of datacenters, ominously referred to as the power wall. In one paper, Google mentions that although it is physically possible to fit more servers in existing datacenters, they cannot do so because this would exceed the power densities that their cooling systems can dissipate [8]. Figure 2.3 shows a breakdown of power consumption in Google's datacenters. It is interesting to note that the most power hungry components of datacenters are the CPUs and the cooling solutions, drawing 42% and 15.4% of total power consumption, respectively.

2.2 Datacenter Applications

We use the term datacenter application to refer to application-level software running in the datacenter that executes a complete user application. TensorFlow [11], a large-scale deep-learning framework, and Pregel [12], a large-scale graph processing framework are both examples of datacenter applications.
Datacenter applications can be broadly classified as being either an online service or an offline job [9]. The former, online services, are characterized as applications that continuously process streams of requests. Examples of online services include web search, Distributed File Systems (DFS), and online maps. Online services are typically customer facing. Thus, these applications operate under tight latency constraints and must be highly reliable. Google, for instance, expects their services to operate correctly 99.99% of the time, which corresponds to only 52 minutes of down time per year [3]. Online services use approximately 60% of all CPU time and account for 85% of all RAM allocations in Google’s datacenters [9]. In contrast to online services, offline jobs are “one-off” computations that resemble super-computing workloads. Examples of offline jobs include MapReduce workloads and large-scale graph processing. Due to the nature of the work and its internal customer base, offline applications are typically latency tolerant. These jobs may last anywhere from a few seconds up to a few days. Offline jobs account for roughly 30% of all CPU time on Google’s clusters [9]. For this thesis, we focus our attention on online services. Section 4.1 presents our rationale for doing so.

Without exception, datacenter applications are distributed – they are composed of many sub-programs executing on many different nodes. One reason is distribution allows
Figure 2.4: Diagrams showing a decentralized (left) and a centralized (right) application.

us to exploit inherent parallelism common to these applications. For example, SQL processing is a popular datacenter application that is distributed primarily for parallel execution. Table join (i.e. FROM clauses) and row selection (i.e. WHERE clauses) are entirely parallel operations. By spreading them over many systems and executing them simultaneously, overall query execution time can be reduced. Applications are also made distributed so that they can continue to operate in the presence of hardware failures and maintain high uptime. By dividing execution across several nodes, lost progress is limited to the work that was executing on the nodes that failed. Still, a fault tolerant application may still cease execution when a critical part of the distributed system fails. Certain applications cannot afford to be down, and are replicated in their entirety as a result. For databases, distribution is necessary for these systems to scale beyond the storage capacities of a single computer. The Google File System (GFS) [13], for instance, is a Unix-like file system that stores its files across many nodes. At the time it was originally proposed in 2003, one GFS instance had already grown to 180 TB, well beyond the limits of current persistent storage solutions. Finally, some applications operate within tight latency constraints. However, if client and server are separated by large geographical distances, the communication becomes noticeable and for some applications, intolerable. This is one of the reasons why the largest datacenter operators distribute their systems geographically.

2.3 Distributed Systems

Distributed systems are applications that execute across many compute nodes. They may be broadly classified as centralized, or de-centralized. As the names suggest, the differentiating factor is the existence of, or lack thereof, a central controller or master in their designs. Figure 2.4 shows a side by side comparison of these two models.
De-centralized systems are characterized as a collection of peer processes, with no distinct master, working together to run a single application. Global state is maintained through consensus protocols such as Paxos [14]. ZooKeeper [15] and Megastore [16] are two examples of commercial deployments of de-centralized-systems. De-centralized systems were invented to overcome the issues that arise when building applications that run across many computers. For one, the task of orchestrating the overall efforts of the distributed system becomes arduous as node membership increases. By de-centralizing control, the burden is spread over all participants. Furthermore, datacenter applications must continue to operate correctly in the presence of frequent software and hardware failures. However, this is somewhat difficult to achieve in centralized systems due to the critical role that the controller plays – the remainder of the system pauses execution in the absence of instruction from the controller.

Fully de-centralized distributed systems seem to have many traits that are attractive to datacenter operators. Yet, despite their boons, datacenter operators tend towards centralized models. The reason is that centralization greatly simplifies implementation and has proven to be adequate to date [3, 8]. In contrast to de-centralized systems, centralized models employ a central controller, or master, to maintain global state and coordinate the system. There are many examples of centralized systems: Web search [8], Map Reduce [17], and TensorFlow [11] to name a few.

Our work focuses on the master-worker model, a popular centralized distribution model, summarized in Figure 2.5. The user application is distributed by dividing it into two kinds of entities: a master, and many workers. The master is analogous to the main() function in C programs. It coordinates the overall effort of running the application. Workers are analogous to regular functions that are invoked by main(). They are engines that continuously receive and service requests to perform a specific piece of work. When work is parallelizable or too burdensome for the master to run, the master offloads it to workers by issuing tasks, analogous to function calls. Tasks are collections of all input parameters and metadata necessary for a worker to execute the work it describes. During operation, masters submit tasks to the network without any knowledge of how many workers exist, their latency, throughput, or current load. It is the responsibility of the datacenter infrastructure to route tasks to workers in a way that best suits the application.
Many of the advantages of using a master-worker model stem from the ability to dynamically scale. That is, master-worker systems can gracefully handle the increase or decrease of workers during execution. When additional workers join the system, the network need only to route tasks to them to leverage these new compute nodes. If a node unexpectedly leaves, perhaps due to a failure, the task that was lost can be re-issued to another worker instance. This is known as fault tolerance through re-execution.

The master-worker model also has the advantage of being able to share workers amongst masters. Any master may commit tasks to any worker via the network. This feature is useful in maximizing the utilization of workers. Consider a simple application that applies the formula $f(x) = 2x + 1$ to each element of an array. Figure 2.6 shows how this service would be implemented as a single-threaded C program. Suppose that we distribute its execution in a master-worker fashion by creating a worker that multiplies all elements of the array by two, and another worker that adds one. The master issues the task to the times two worker, and upon receiving a response, issues another task to the add one worker. Notice that there is always one worker that is idle. This problem may be remedied by adding a second master that services another request to apply $f(x) = 2x + 1$ to all elements of a different array. Both masters commit work simultaneously, causing both workers to be used all the time. Worker sharing is elaborated upon in Section 4.3.2.
const int SIZE = 100000;

void times_two_add_one (int array_of_int[SIZE], int n) {
    for (int i = 0; i < n; i++) {
        array_of_int[i] = 2 * array_of_int[i] + 1;
    }
}

Figure 2.6: Implementation of applying the \( f(x) = 2x + 1 \) to all elements of an array

Master-worker should not be confused with master-slave. In the master-slave model, the master is aware of all participating slaves, and assigns work accordingly. In contrast, the master in the master-worker model issues tasks irrespective of how many workers there may be to run those tasks. Master-worker is preferred over master-slave at scale for its ability to dynamically scale.

\section*{2.4 High-Level Synthesis}

Our work enhances the HLS programming model to be able to describe master-worker distributed systems. We briefly comment on what HLS tools are and the programming model they follow.

HLS tools are a class of Computer Aided Design (CAD) tools that convert software programs into hardware blocks implementing the programs. The emergence of these tools in the 1990s was driven by improvements in process technologies \[18\]. As more transistors were being added to the silicon die, hardware designers were building bigger, more complex systems. Design, testing, and evaluation of RTL written in HDLs was becoming burdensome. Increased design abstraction was clearly necessary. Today, advancements in HLS have made it a viable means of building high-quality hardware more quickly than hand writing RTL in HDLs. Both major FPGA vendors offer an HLS solution – Intel created A++ \[19\], while Xilinx has Vivado HLS \[5\]. Products like Altera OpenCL \[20\] and SDAccel \[21\] further bridge the gap between software and hardware development by enabling users to leverage the FPGA through Open Compute Library, a programming model familiar to developers who accelerate their programs using General Purpose Graphics Processing Units.

To use HLS, users provide a software program, typically written in C/C++, and name a \textit{top-level function}. The top-level is what would normally be called the entry point
into the program. It is used by HLS to determine the structure of the program and what sub-functions are relevant to synthesis. Additionally, the engineer will likely annotate constructs within the program, such as variables, functions, and loops with HLS pragmas. These pragmas enable users to make macroscopic changes to the hardware generated. For instance, one can instruct Vivado HLS to implement a block of code as a hardware pipeline instead of a datapath and FSM using the DATAFLOW and PIPELINE directives. As another example, one can choose whether to implement multiplication via DSP blocks or in LUTs using the RESOURCE pragma. It is important to note that while these pragmas are expressive, and HLS tools are fairly powerful, they do not have any notion of distribution.
Chapter 3

Related Work

In this section, we survey and distinguish our work from prior art.

The endeavor of building tools and programming models that make programming distributed systems easier is by no means a new field of study. Some work dates as far back as the 1980’s, at the advent of supercomputers. At the time, these systems were large clusters of CPUs, not unlike today’s datacenters in concept. While some of the ideas and algorithms that have been proposed over the years bear some resemblance to our work, our work differs fundamentally in the applications it targets. As a consequence of concentrating on super-computers, many of the design decisions made by prior art are ill-suited for the datacenter and its applications, that we focus on.

3.1 OpenMP

The Open Multi-Processing project (OpenMP) [22] is a set of pragma-based extensions to the C programming language that enable developers to compile single-threaded programs into multi-threaded executables. The OpenMP initiative was borne out of the desire to easily build applications that fully leverage multi-core processors, such as Intel’s Xeon and Celeron processors, which were gaining traction at the time. Its design reflects this target platform.

The primary parallelization mechanism in OpenMP is the parallel directive. It denotes that the code block following it should be run by one or more threads. In the example shown in Figure 3.1, “Hello World!” will be printed at least once, and then once more for each additional thread spawned. The number of spawned threads, among other parallelization parameters, is chosen during execution by the OpenMP runtime, a dedicated
int main(void) {
    #pragma omp parallel
    {
        printf("Hello World!\n");
    }
    return 0;
}

Figure 3.1: A trivial example demonstrating the use of the parallel pragma in OpenMP. “Hello World!” will be printed one or more times, once for each spawned thread.

thread that spawns and manages OpenMP threads. The runtime can be guided by other OpenMP pragma and pragma parameters. When all participating threads have printed “Hello World!”, serial execution continues. In other words, OpenMP follows a fork-join multi-tasking model.

Worksharing directives are the most similar constructs to our programming model. Used in tandem with the parallel pragma, this class of directives denote that execution of a block of code should be spread across spawned threads, rather than re-executed verbatim. The for pragma, for instance, denotes that iterations of a for loop should be split across spawned threads. The example code shown in Figure 3.2 will always print “Hello World!” ten times, regardless of the number of threads the OpenMP runtime may spawn. In the absence of the for pragma, “Hello World!” will be printed at least ten times, and ten times more for each additional thread spawned.

OpenMP’s programming model is like ours in that both allow the developer to express distribution of work and data across threads. Our work differs in the distribution and memory models of the entities that are generated. OpenMP follows a master-slave, shared memory model – the master and its slaves share the same address space and can directly access each other’s memory. On the other hand, our work generates IP blocks that interact in a master-worker relationship, and follow a distributed memory model. It is these key differences that make our solution better suited for the datacenter and its applications. Shared memory systems are highly undesirable in datacenter environments because they generate too much unproductive traffic [8]. Data writes must be broadcast to all participants regardless of if the information is used. As was detailed in Section 2.3, the master-worker distribution model is preferred to the master-slave model because it provides dynamic scalability, load-balancing in the face of heterogeneous hardware resources, worker sharing, and lends itself naturally to checkpointing.
int main(void) {
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 10; i++) {
            printf("Hello World!\n");
        }
    }
    return 0;
}

Figure 3.2: Example usage of the for directive. “Hello World!” will be printed exactly ten times.

There has been work on supporting OpenMP for HLS tools [23, 24]. These tools focus on driving performance by implementing parallel child threads as hardware accelerators resident on the FPGA portion of a CPU-FPGA system. In addition to the disadvantages of OpenMP that disqualify these works for use in the datacenter, they do not seem to support worksharing constructs.

3.2 Distributed Memory Compilers

Distributed Memory Compilers (DMC) are tools that compile a sequential program onto a target distributed memory computer, given its dimensions (i.e., network topology, CPU specifications, memory size, etc.). It treats the cluster as a single logical computer, planning the optimal execution plan and distribution of data among its nodes. These tools saw a surge in research in the 1980’s and 1990’s motivated by growing demand for scientific applications and the inability of shared-memory machines to scale to meet these needs. PARADIGM [25] and the SUPERB compiler [26] are two examples of DMCs.

Many of the challenges that DMCs aim to address are similar to our own. Both our work and DMCs inject communication logic and divide execution of a sequential program across many nodes. Where our work differs is the distribution model. DMC’s create execution plans for a sequential program, injecting logic to transmit data between nodes at specific points in the code. Effectively, DMC’s create application specific, fully de-centralized, distributed systems. By creating master-worker systems instead, the distributed systems built from the IP blocks our work generates can scale dynamically, and
are easily made fault tolerant. As we argued throughout Chapter 2, these are important considerations when building datacenter applications. Systems created using DMCs, in contrast, must be re-compiled every time the cluster changes in size.

3.3 FPGAs in Datacenters

Research on enabling FPGAs in datacenters has seen a sharp increase after Microsoft published their work on Catapult [6]. Most work views the FPGA as a low-power co-processor that must be connected to a CPU [6]. Other work has argued that FPGAs ought to be standalone, and directly attached to the datacenter interconnect [27, 10]. There are also ongoing efforts to virtualize the FPGA [27, 28], that is, enabling their support within virtual machines and Infrastructure as a Service (IaaS) platforms such as Elastic Cloud 2 (EC2). Our work is orthogonal to these branches of literature. Our work and our vision aim to enable the implementation of datacenter applications entirely on FPGAs, independent of whether the FPGA is virtualized or connected to a host CPU.
Chapter 4

High-Level Synthesis of Datacenter Services

In this chapter, we present our proposed programming model enhancement to HLS tools.

4.1 The Case for Datacenter Services on FPGAs

Datacenters must continue to scale to meet growing demand for datacenter-based applications. However, power consumption becomes debilitating when building larger datacenters. Energy is expensive and cooling solutions are already at their limits today. To continue to add performance to the datacenter while operating within tight power budgets, we propose and envision that entire online datacenter services will be implemented on FPGAs. We distance ourselves from the view that FPGAs are accelerators that must be controlled by a host CPU. Instead, we see the FPGA simultaneously as a low-power processor, a parallel processor, and an I/O controller [29].

Online services appear to be good candidates for total implementation on FPGAs. First, there seems to be significant power saving potential. Consider the following: In Google’s datacenters, CPUs account for 42% of all datacenter power draw [3] and online services account for 60% of all CPU time [9]. Therefore, 25% of all datacenter power is used on CPUs running datacenter services.

\[
\text{% DC power for CPU} = 42\% \quad (4.1)
\]

\[
\text{% CPU time for services} = 60\% \quad (4.2)
\]
% DC power for CPUs running services = 60% × 42% = 25%  \hspace{1cm} (4.3)

The Stratix V used in Microsoft Catapult uses under 25 W of power [6], approximately 15% the power of current generation, 165 W CPUs [30] used in datacenters.

Stratix V power = 25 W \hspace{1cm} (4.4)

Xeon power = 165 W \hspace{1cm} (4.5)

\[
\frac{\text{Stratix V power}}{\text{Xeon power}} \times 100\% = \frac{25 \text{ W}}{165 \text{ W}} \times 100\% = 15\%
\hspace{1cm} (4.6)
\]

If datacenter services are entirely implemented on FPGAs instead of CPUs, the power used to run these services can be reduced to 15% of its original value. We arrive at the conclusion that the total datacenter power used to run datacenter services can be reduced to approximately 4% of current datacenter power draw, a power savings of about 21%. Although these calculations are optimistic approximations, they show that there is some possibility of significant power savings worth further investigation.

Note that our calculations here are a rough estimation. A more comprehensive calculation is necessary to determine whether it would be economically beneficial to implement services strictly on FPGAs. First, one would need to ensure that the performance of service implemented strictly on FPGAs performs at least as well as a CPU only solution. Second, the total cost of ownership of an FPGA enabled datacenter would need to be compared to that of a CPU-only datacenter. As a point of reference, Microsoft’s FPGA accelerated ranking of Bing search results increased performance by 95% at under 30% increase in TCO and under 10% power increase [6]. This is strong evidence that FPGAs can address both the performance and power issues, but only for one application.

% DC power for datacenter services using FPGAs = 25% × 15% ≈ 4%  \hspace{1cm} (4.7)

% DC power saved by converting to FPGAs = 25% – 4% ≈ 21%  \hspace{1cm} (4.8)

Second, online services have infinite lifetimes [9]. Applications implemented in hardware should have long lifetimes so that the performance benefits and power savings resulting from a hardware implementation eclipse the additional development cost. Offline jobs, in comparison, may not be worthwhile for implementation on FPGAs since they run for at most a few days [9]. The cost of building a hardware implementation is amortized over a much shorter time.
Figure 4.1: Simple implementation of adding one to every element to an input array

Third, datacenter applications are updated on a weekly basis [3]. This regular update cycle is short enough to make ASICs far from viable, but long enough to greatly overshadow the time to reconfigure FPGAs.

Fourth, online services exhibit massive request-level and data-level parallelism. FPGAs lend themselves well to such problems, since parallelism is easily leveraged by replicating IP blocks and functional units within the blocks themselves.

In summary, datacenter services present an interesting opportunity for power savings by implementing them on FPGAs. The constraints associated with developing on FPGAs matches the characteristics of datacenter services.

4.2 Programming Model

In Sections 2.2 and 4.1, we discussed online services and argued that they are good candidates for implementation on FPGAs. Section 2.4 describes HLS, a hardware design methodology growing in popularity. We now present our contribution, which bridges the gap between the applications that we believe ought to be implemented on FPGAs, and the tools that engineers use to create FPGA-based hardware.

Consider the Vivado HLS C code shown in Figure 4.1 that does the simple operation of adding one to every element in a large array. Suppose that we wish to implement it as a datacenter service, distributing the work across a number of worker nodes as shown in Figure 4.2. The master partitions the data and issues requests to process each portion of the array. Workers execute tasks and reply to the master with the updated data. The master aggregates the partial results and returns the array to the requester with each element incremented by one. Clearly, there is some discrepancy between the code shown in Figure 4.1 and the architecture we wish to actualize.
Figure 4.2: Block diagram showing the master-worker implementation of the add one application in Figure 4.1.

To easily build hardware cores for the master and workers, we propose that HLS tools be augmented to be able to express master-worker distributed systems in their programming models. We introduce a new pragma, DISTRIBUTE, to a production HLS tool, Vivado HLS, and provide prototype support for it. The DISTRIBUTE pragma is a function-level annotation that is used to declare that a sub-function (i.e., a non top-level function) is to be implemented as a discrete hardware engine. We refer to sub-functions annotated with the DISTRIBUTE pragma as distributed functions, their hardware implementations as workers, and their callers as masters. All generated hardware blocks participating in the distributed system, including the top-level, all masters, and all workers, are called distributed blocks.

Invocations to distributed functions are analogous to synchronous remote procedure calls, except implemented in hardware. When a master “invokes” a worker, it packages parameters along with metadata and issues a request. Workers continuously accept requests, process them, and issue responses containing the results. Masters pause execution until a response is received for its request.

The flow of data between masters and workers follows the C programming model. Function parameters and return values are interpreted as data that are exchanged between caller and callee, and are consequently communicated between master and worker. Only parameters that are read are transmitted from master to worker, and only parameters that are written are sent from worker to master. Global variables, on the other hand, are
Figure 4.3: Example callgraph demonstrating vertical scaling. `fooA()` is the top level function. `fooB()` and `fooC()` are distributed functions. Both `fooA()` and `fooB()` will generate request messages when “invoking” `fooC()`.

treated as data that are privately managed by each distributed block. Although static local variables are an exact fit for this role, we chose to use global variables since Vivado HLS offers greater flexibility with global variables, allowing them to be backed by a user defined memory such as off-chip DDR memory or a hard disk.

Like in software, distributed functions may invoke one another to decompose large problems into smaller ones, or to factor out common functionality. For example, the callgraph shown in Figure 4.3 shows a top level function, `fooA()`, that outsources part of its functionality to `fooB()` and `fooC()`. Similarly, `fooB()` also invokes `fooC()` to do part of its work. Had these been distributed functions, distributed blocks would be generated for `fooA()`, `fooB()`, and `fooC()`. By decomposing `fooA()` into `fooB()` and `fooC()`, the distributed block for `fooA()` is made smaller since it implements less functionality. Furthermore, `fooC()` instances may be shared between `fooA()` and `fooB()` blocks, rather than instantiating separate `fooC()` distributed blocks for `fooA()` and `fooB()`. In this example, `fooB()` is a master to `fooC()`, but is a slave to `fooA()` – masters may themselves be slaves to other masters.

A key feature of the DISTRIBUTE pragma is the ability to automatically parallelize execution of distributed functions along one or more arrays that the user has specified. The code in Figure 4.1, for instance, can be parallelized by partitioning `array_of_int` and adjusting the for loop so that it operates on a portion of the array. This way, each section of `array_of_int` can be processed simultaneously by multiple add one worker instances, as was shown in Figure 4.2. When this feature is enabled, the master splits the named arrays
Figure 4.4: The syntax of the DISTRIBUTE pragma. When a function is labelled with the DISTRIBUTE pragma, it is implemented as a separate IP core. The user may optionally specify the partition size and elements variable if they wish to partition execution of the function.

```c
void foo(...) {
  #pragma HLS DISTRIBUTE
  [partition_size=<elements constant> elements=<imm32>]
  // function body
}
```

Figure 4.5: Example usage of the DISTRIBUTE pragma, applied to the add one example shown in Figure 4.1.

```c
const int SIZE = 10000;

void add_one(int n, int array_of_int[SIZE]) {
  #pragma HLS DISTRIBUTE partition_size=100 elements=SIZE
  for (int i = 0; i < n; i++) {
    array_of_int[i]++;
  }
}

void top_level(int n, int array_of_int[SIZE]) {
  add_one(n, array_of_int);
}
```

and issues one task for each partition. Logic is injected into the master to aggregate the results that are received in response to the issued tasks. If functional equivalency cannot be maintained, this feature should cause a compile-time error.

### 4.2.1 Pragma Syntax

The DISTRIBUTE pragma is used to mark a sub-function for conversion into a discrete engine: a hardware block that continuously accepts and processes parameter vectors. Its syntax is given in Figure 4.4, and Figure 4.5 shows the DISTRIBUTE pragma applied to our add one example. The result of running an HLS flow that supports DISTRIBUTE should be IP cores for the top-level function and each distributed function.

To enable automatic partitioning, the user must specify both a partition size and an elements constant. The HLS tool partitions all arrays in the distributed function that use
the elements constant in its size expression as part of its declaration. At compile-time, the HLS tool resizes the partitioned arrays by substituting the elements constant with the partition size, an immediate integer value. The partitioned arrays may be a function parameters, global or local variables, or the function return value.

### 4.2.2 Generated IP Cores

Discrete hardware blocks are generated for the top-level and each distributed function. Each worker is assigned a non-intersecting range of *worker kind IDs*. A worker kind ID is a whole number representing a group of workers that implement the same distributed function, and share persistent state. Namely, the memories that are used to implement global variables accessed by the workers must be coherent with each other. Note that sharing the same memory for all worker instances belonging to a single worker kind ID is a legal configuration. For the code shown in Figure 4.1, a single worker kind ID, such as 1234, is sufficient to group all add one worker instances – they all share the same persistent state (i.e., no persistent state), and implement the same distributed function.

During operation, distributed blocks will exchange messages with each other. Messages will either be directed at a distributed block instance or towards a worker kind ID. In the case of the latter, the network must forward the message to exactly one worker instance belonging to that worker kind ID. The load-balancing function that routes packets to a worker instance is left to the user’s discretion. One may, for example, choose to use a round-robin load-balancer for simplicity. In practice, load-balancers are complex functions of many variables such as the latency of worker nodes, their current load, and the priority of the work.

At minimum, each distributed block will have an ID port, and a pair of input/output ports used to exchange messages with other distributed blocks. Figure 4.6 shows an example of a distributed block. The communication ports are implemented using the Advanced eXtensible Interface (AXI) stream protocol [31], a bus protocol supported by Vivado HLS that is optimized for packetized communication. Care must be taken to obey the TDEST and TLAST signals that indicate packet destination and boundaries, respectively. All data that appears in the parameter list and the function return value, are transmitted in the order they appear. By doing this, the generated communication protocol for a distributed function remains the same across different applications that use the distributed function. This feature, which we refer to as worker sharing, is explained in detail in Section 4.3.2.
The ID port is used to configure IP cores with a globally unique ID (GUID). It is used during transmission to refer to a specific distributed block rather than a worker kind ID grouping. The 32-bit value fed into the ID port must remain stable while the IP core is running and must not intersect with any worker kind ID ranges. Users will likely populate this port with a constant value (i.e., hardwire the port) or connect it to an AXI attached register.

In addition to the AXI stream and ID ports, a new master AXI port is generated for each global variable accessed by the distributed function. It is the engineer’s prerogative as to what memory to back the master AXI port with. For instance, one may opt to use a hard disk for its size and persistency, or LUTRAM for its speed. At the very least, the memory must be sufficiently large to hold the data belonging to the global variable that the master AXI port implements.

4.2.3 User Guarantees

In this section, we enumerate all assumptions made by hardware that must be upheld by the user to ensure correct operation. The first assumption is specific to our implementation, but the remaining two are fundamental to the programming model.

Exactly once, error-free delivery All packets must be delivered exactly once to their
destination, free of errors. Packets may be delivered out of order. If the transmission medium is unreliable, one may need to wrap all traffic in a protocol that provides reliability guarantees (e.g. TCP) to satisfy this requirement.

**Worker kind anycast** When a packet’s destination is set to a worker kind ID, the user must forward this packet to exactly one instance of a worker belonging to that worker kind. One possible method of accomplishing this is by routing all traffic sent to a worker kind ID to a load balancer that replaces the TDEST field with the GUID of a worker instance.

**Uniqueness of GUIDs** GUIDs assigned by the user must be unique within the distributed system being built using the distributed blocks. GUIDs must not collide with any worker kind IDs.

### 4.3 Using the IP Cores

In this section, we demonstrate the expressiveness of our programming model and how to build scalable master-worker distributed systems using the distributed blocks it describes.

To facilitate our discussion, we revisit the example presented in Figure 2.6 that applies the formula \( f(x) = 2x + 1 \) to all elements of an input array. For illustrative purposes, we break the computation into two stages, `times_two()` and `add_one()`, and create a top-level that invokes these functions. The refactored code is shown in Figure 4.7. Both subfunctions are labelled identically with the `DISTRIBUTE` pragma, declaring that they should be implemented as discrete IP cores. We name `SIZE` as the elements variable, and that the generated workers should behave as though `SIZE=1000` by setting partition size to 1000.

Given this code, our system generates distributed blocks corresponding to `top_level()`, and the engines implementing `times_two()` and `add_one()`. Because neither `times_two()` or `add_one()` are stateful, each of these distributed functions are only assigned a single worker kind ID. Let us assume these are 0 and 1, respectively.

During operation, the master, `top_level()`, first executes `times_two()`. It divides `array_of_int` into \( \frac{\text{elements}}{\text{partition size}} = \frac{100000}{1000} = 100 \) chunks. To form complete tasks, it prepends its own ID, a task index, and the \( n \) parameter. The master then transmits these tasks...
const int SIZE = 100000;

void times_two(int array_of_int [SIZE], int n) {
#pragma HLS DISTRIBUTE elements.var=SIZE partition_size=1000
    for (int i = 0; i < n; i++) {
        array_of_int [i] *= 2;
    }
}

void add_one(int array_of_int [SIZE], int n) {
#pragma HLS DISTRIBUTE elements.var=SIZE partition_size=1000
    for (int i = 0; i < n; i++) {
        array_of_int [i] ++;
    }
}

void top_level(int array_of_int [SIZE], int n) {
    times_two (array_of_int, n);
    add_one (array_of_int, n);
}

Figure 4.7: Distributed implementation of times two add one.

on its outbound AXI stream port, and pauses execution while it waits for responses for its tasks. When the updated data arrives, it follows a similar procedure to execute add_one().

4.3.1 Worker Replication

We begin our discussion by demonstrating scaling of our fictitious service from one worker instance to many. It is often the case that replicating workers is desirable. First, computation is parallelized over many workers, reducing latency. Recall that latency is of critical importance to online services, since they are typically customer facing. For distributed storages where workers hold partitions of the database, replication provides reliability and availability through redundancy, and allows the system’s storage to scale well beyond the limits of a single node.

We first demonstrate a small-scale deployment of our distributed blocks. One instance of each distributed block is instantiated and connected as shown in Figure 4.8. The outbound AXI stream port of top_level() is connected to a demultiplexer that routes
Figure 4.8: Block diagram of small scale implementation of \( f(x) = 2x + 1 \). top_level is connected to one instance of add_one and times_two using a multiplexer and a demultiplexer.

packets to the times_two() or add_one() workers based on the TDEST field. Response packets from the worker cores are consolidated into a single stream that is then forwarded back to top_level(). Note that execution is completely serialized in this configuration. Since only one of each worker kind exists, only one of each kind of task can be processed at a time. Although slow, this configuration is legal and in fact valuable. The ability to build miniature versions of services is useful during the development process. Rather than having to deploy a design at full-scale to test it, engineers can build a small-scale version locally, potentially using only one FPGA. The simplicity of this configuration eliminates points of failure that are extraneous to the application itself, keeping development time focused. For example, there is no need for a load-balancer, scalable interconnect, or communication between FPGAs, all of which may contribute bugs that slow development and distract from the application logic.

We improve the performance of our distributed system by scaling the number of worker instances. Figure 4.9 shows our new configuration and Figure 4.10 summarizes the flow of packets. For brevity, we represent the AXIS_in and AXIS_out ports as a single AXIS_in/out port. An arbitrary number of add_one() worker instances along with a hand-written, round-robin load-balancer that is aware of the GUIDs of all participating workers have been added to the design. All distributed blocks and the load-balancer are connected using an AXI stream packet switch, an interconnect that can better handle the increased connectivity and traffic than a multiplexer/demultiplexer pair, while using fewer resources. The packet switch is configured to route all anycast traffic – packets that set TDEST to either 0 or 1 – to the load-balancer. The load-balancer replaces incoming packets’ TDEST field with the GUID of a worker and returns the packets to the switch.
Here, requests simultaneously run across many workers, reducing execution time.

### 4.3.2 Worker Sharing

It is highly desirable to maximize utilization of all available resources in datacenter environments. By doing so, less hardware is needed to run the same set of applications. FPGAs, if introduced to the datacenter, will be no exception to this rule. We remind the reader of the problem, first presented in Section 2.3, that arises when building applications with serial components. The sample application that applies $f(x) = 2x + 1$ to all elements of an array has two phases of execution. First, it multiplies all elements by two, and then it adds one to each element of the array. At any time, either the `times_two()` or `add_one()` workers are idle. This issue may be addressed by replicating the master, `top_level()`. Multiple requests to apply $f(x) = 2x + 1$ to different arrays may be serviced simultaneously and contribute work that keeps all worker nodes busy, as shown in Figure 4.11. Multiple `top_level()` instances commit work to the scheduler that load balances the requests onto `add_one()` and `times_two()` instances. To build a functioning distributed system, one need only to add more `top_level()` instances, as shown in Figure 4.12.
Figure 4.10: Diagram showing the flow of packets with one instance of `top_level` and multiple instances of `add_one` and `times_two`. `top_level` outputs a stream of requests that are routed to the scheduler. The scheduler load-balances incoming requests to the `add_one` and `times_two` workers.

Figure 4.11: Diagram showing multiple `top_level` instances committing work to multiple `add_one` and `times_two` instances. The scheduler accepts requests from all instances of `top_level`. It load-balances the requests to instances of `add_one` and `times_two`. The workers directly communicate their results back to the original requesting `top_level` instance.
Figure 4.12: Block diagram showing multiple top_level instances pushing work to multiple times_two and add_n instances.

The concept of worker sharing can also be applied across applications. Consider a separate application that applies the formula $f(x) = 2(x + 1)$ to all elements of an array. The implementation of this application is identical to the code shown in Figure 4.7 that implements $f(x) = 2x + 1$, except that the invocations to add_one() and times_two() by top_level() are swapped. In other words, the add_one() and times_two() workers for both applications are identical, and can be shared between these applications. Let the top-level function of the application implementing $f(x) = 2(x + 1)$ be called top_level_new(). Figure 4.13 shows how these applications can share workers, thereby boosting utilization.

4.3.3 Deployment in the Datacenter

The goal of this work is to provide the building blocks of a master-worker distributed system that implements the core functionality of a user’s application expressed as a single-threaded program (Vivado HLS C). For distributed blocks to be used in the datacenter, as is our vision, cluster management systems, the tools that handle the complexities of deployment in a datacenter, must be augmented to include FPGAs. These tools must be
able to provision entire or parts of FPGAs as they are needed, and reclaim them when they are not. They must be enabled to instantiate distributed blocks onto reserved FPGAs, and configure the network to correctly forward traffic between distributed blocks. Finally, they must be enhanced to be able to detect hardware failures and take appropriate action, perhaps by asserting the affected distributed blocks’ reset signal or by re-instantiating them on other FPGAs. Examples of cluster management systems include Borg [9], Fuxi [32], and Apollo [33].

There are ongoing efforts at the University of Toronto to enhance OpenStack, an open source cluster management system, to handle heterogeneous datacenters – those with resources other than CPUs such as Graphics Processing Units (GPUs), and FPGAs. Users provide OpenStack with the hardware blocks, called kernels, in their distributed systems and specify connections between kernels using a directed graph. The user must also specify the partitioning of kernels onto FPGAs. Given the aforementioned parameters, work by Tarafdar et al. [34] deploys the system to the datacenter. This entails reserving the necessary FPGA resources, instantiating the hardware blocks, and connecting the blocks according to the directed graph. The system may optionally be replicated as a whole, and incoming work automatically load-balanced onto an instance.

Our work complements that are described in [34]. While ours takes a user application and provides the distributed blocks, the system in [34] takes the necessary steps to fully deploy the cores into a working distributed system.
4.3.4 Building Large Scale Distributed Systems

Distributed blocks are not constrained to coexist on a single FPGA. On the contrary, we designed our distributed blocks so that they may exist on any number of FPGAs and communicate using any medium. Figure 4.14 shows how a distributed application can be implemented over multiple FPGAs spanning the Internet. The ability to scale applications across multiple FPGAs is an important feature. Applications can scale to meet Internet-scale demand, and there is no single point of failure. Each FPGA houses some arbitrary combination of distributed blocks. This is a desirable property for datacenter operators who minimize procurement costs by maximizing utilization of existing hardware. Distributed blocks can be combined to match the resource ratio of the underlying FPGA, maximizing utilization.

4.4 Examples

In this section, we present examples of how real datacenter services can be expressed using our model. The objective of this section is to demonstrate to the reader that real world applications can be expressed using our programming model and built using the distributed blocks that are generated. We make simplifications where necessary to keep our discussion focused on this goal.

4.4.1 Web Search
Web search is the quintessential datacenter service. It maps a set of query words to a set of web pages that contain those words. Today, web search has become an elaborate set of algorithms working in tandem to produce the best possible search results. To understand how web search can be implemented using our programming model, it is helpful to first establish the overall web search algorithm. Web search has four stages of execution, shown in Figure 4.15, that we refer to as Reverse Index Look Up, Document ID Intersection, Ranking, and URL Translation.

The first stage, Reverse Index Look Up, is responsible for mapping each query word to a set of document IDs that represent matching web pages. Document IDs are whole numbers that serve as a concise way of referring to web pages – the number 7149 can be used instead of www.reallyLongUrlsAreDifficultToWorkWith.com to save memory during intermediate computations. Translation of query words to matching document IDs is done by consulting a pre-computed perfect hash table, called the reverse index. The reverse index uses query words as keys and returns sets of matching document IDs. For example, the query “Cat Videos” is composed of the query words “Cat” and “Videos”. Each of these words are used to lookup document ID matches in the reverse index – “Cat” may match with the document ID set \{1,2,3,4\}, while “Videos” may result in the set \{2,3,4,5\}. Sets of document IDs that match only one of the query words are referred to here as word matches.

The Document ID Intersection phase determines a set of document IDs that match all input query words, which we call a query match. Query matches are computed by
determining the set intersection of all word matches, forming a new set of document IDs that exist in all word matches. As an example, the intersection of the word matches for “Cat” and “Videos” is \( \{1, 2, 3, 4\} \cap \{2, 3, 4, 5\} = \{2, 3, 4\} \). Note that it is possible for the query match to be an empty set, denoting that no web pages contain all of the input query words.

The goal of the Ranking stage is to order the query match in descending order of relevance to the end user. That is, the user should most likely follow the first handful of URL links. Production-grade ranking engines are tightly kept secrets that take into consideration many factors such as current events in the world, user behaviour, and geographical origin of the query, to produce the best possible ranking. However, past work on Google’s ranking algorithm, PageRank, is available for reference [35]. Continuing with the example given in Figure 4.15, the ranking example may choose to reorder the query match set \( \{2, 3, 4\} \) into \( \{3, 2, 4\} \). We omit Ranking from further discussion as a simplification.

The final stage of web search formats the results for presentation to the end user. Normally, this consists of converting each document ID into a URL, and building summaries for each web page. For simplicity, we consider only the former operation of mapping document IDs to the URLs they stood for. This is done by querying a second perfect hash table, which we refer to as a URLs Table, that takes document IDs as keys and produces URL strings. In the example given in Figure 4.15, the document ID ‘2’ may represent “www.d6v.com”, while ‘4’ may map to the URL “www.g3.com”.

The Google search engine implements web search as shown in Figure 4.16. It is structured as three major entities: Google Web Servers (GWS), Index Servers, and Document Servers. GWSs are responsible for coordinating the effort of fully servicing requests. They consult the Index Servers to execute Reverse Index Look Up and Document ID Intersection, and defer to the Document Servers to do URL Translation. The Index Servers are organized into clusters, and each cluster is assigned a reverse index shard. Reverse index shards determine which document IDs belonging to a pre-determined, non-intersecting subset of the set of all possible document IDs match a given query word. They are structured identically to the reverse index: each is a pre-computed perfect hash table that maps query words to some set of document IDs. For instance, a reverse index shard that is responsible for the document IDs \( \{1, 6, 7, 20\} \) outputs subsets of these document IDs that match with an input query word. Each cluster computes the query
match of its assigned document IDs and the resulting sets are unioned by the requesting GWS to produce a complete query match. Document Servers are organized identically to the Index Servers – they are clustered and given responsibility over a shard of the URLs table. When GWS receives the matching URLs from the Document Servers, it packages the result and sends it to the user.

Google may be expressed using our programming model as shown in Figure 4.17. The program contains three functions corresponding to the core constituents of Google web search. The top-level function of the project is the Google Web Server, implemented by the function \( g\_web\_srvr() \). It takes as input the search query and outputs matching URLs through the \( rslt \) parameter. GWS offloads work to the index and document servers by invoking their implementations \( idx\_srvr() \) and \( doc\_srvr() \), respectively.

Our pseudo-implementation of the Index Server, mimicking Google’s, is responsible for carrying out Reverse Index Look Up and Document ID Intersection. It does this by retrieving a word match for each query word, and computing the set intersection with a partial query match. The function has been marked as distributed to denote that \( idx\_srvr() \) should be implemented as a discrete engine. Furthermore, automatic partitioning has been enabled by naming \( MAX\_DOC\_IDS \) as the elements variable and specify
that the partition size should be 250. If \( MAX_{DOC\_IDS} \) is set to 1000, each array whose size is affected by \( MAX_{DOC\_IDS} \) will be partitioned \( \frac{\text{elements constant}}{\text{partitionsize}} = \frac{1000}{250} = 4 \) ways and a block of four worker kind IDs will be assigned. To build a valid distributed system, at least four instances of \( idx\_srvr() \) must be created, each servicing its own portion of the partitioned array(s). Note that the reverse index exists as the global variable \( revIdx \) to denote that this data exists worker-side and is not transmitted between GWS and the Index Servers. On the other hand, the query \( ( qry ) \), number of query words \( ( numQry\_Wrd ) \), matching document IDs \( ( rslt ) \), and number of matching documents \( ( numRslt ) \) are listed in the parameter list of \( idx\_server() \) to denote that they are communicated between \( g\_web\_srvr() \) and \( idx\_srvr() \).

The Document Server, implemented by \( doc\_srvr() \), executes URL Translation by looking up each query word in the URLs Table. The function is annotated with the DISTRIBUTED pragma, identically to the Index Server function. Once again assuming a value of 1000 for \( MAX_{DOC\_IDS} \), four partitions of urlsTab will be created, and a worker kind ID range of length 4 will be assigned to the Document Server. Like Index Server, there must be at least four instances of the Index Server distributed block for correctness – at least one block must be dedicated to servicing each of the four partitions of the URLs Table.

### 4.4.2 FlyWheel

Flywheel is a Hyper Text Transmission Protocol (HTTP) proxy service that compresses web pages prior to transmission [36]. It is an opt-in service intended to reduce bandwidth utilization for mobile devices using roaming data technologies such as 4G/LTE. When an HTTP request arrives, it is routed to a proxy server instance. The proxy is analogous to GWS. It is the master that coordinates the overall effort of servicing requests, in this case fetching and compressing web pages. The proxy begins by obtaining a copy of the requested web page by querying another, standalone service called the fetch service. Note that the fetch service is not part of FlyWheel and is used by other services. After the fetch service responds with the web page, proxy decomposes the web page into its components and routes each to the appropriate compression engine. For example, text is pushed through a GZip engine, and images are transcoded to the WebP format. Finally, the proxy transmits the compressed web page to the end-user’s device, where it is decompressed for viewing. For a real-life workload, FlyWheel is able to compress web pages to 42% of their original size, reducing data usage by 58% [36]. Figure 4.18 details the architecture of FlyWheel.
unsigned revIdx [MAX_WORDS][MAX_DOC_IDS];

void idx_srvr(qry[], numQryWrd, docIds[], *numDocIds) {
    #pragma HLS DISTRIBUTE elements=MAX_DOC_IDS partition.size=250
    for(int i = 0; i < numQryWrd; i++){
        int idx = hash1(qry[i]);
        if(i == 0)
            memcpy(docIds, revIdx[idx], MAX_DOC_IDS);
        else
            intersect(docIds, numDocIds, revIdx[idx]);
    }
}

char urlsTab [MAX_DOC_IDS * MAX_URL_LEN];
void doc_srvr(docIDs[], numDocIds, urls[][]) {
    #pragma HLS DISTRIBUTE elements=MAX_DOC_IDS partition.size=250
    for(int i = 0; i < numDocIds; i++){
        int idx = hash2(docIds[i]);
        memcpy(urls + i, urlsTab + idx, MAX_URL_LEN);
    }
}

void web_search(qry[], numQryWrd, rslt[][], numRslt){
    idx_srvr(qry, numQryWrd, docIDs, &numRslt);
    doc_srvr(docIDs, numRslt, rslt);
}

Figure 4.17: Code showing the basic structure of Google Web Search implemented using the DISTRIBUTE pragma.
Figure 4.18: Here, a block diagram of FlyWheel is shown. The fetch service is separate from FlyWheel and used by other services.

Figure 4.19 provides pseudocode for how this service may be expressed using our programming model. Our simplified implementation performs the basic operations of FlyWheel: a web page, obtained via an external service, is broken into its components and compressed using worker engines. The top-level function, \texttt{proxy()}, takes the requested URL as input and outputs a bitstream representing the compressed web page. To query the external fetch service, \texttt{proxy()} invokes the stubbed, distributed function, \texttt{fetch()}. The \texttt{fetch()} function has no body, and is labelled as distributed with automatic parallelization disabled (no elements variable and partition size specified). This causes the distributed block that implements \texttt{proxy()} to generate a task to query \texttt{fetch()}, and expect a response. The distributed block implementing \texttt{fetch()} should be discarded and the tasks routed to the fetch service. The remainder of \texttt{proxy()} iterates over each component of the web page and invokes the appropriate compression worker (i.e., \texttt{compress\_txt()}, \texttt{compress\_audio()}, \texttt{compress\_img()}). Like \texttt{fetch()}, each compression worker is marked as distributed without automatic parallelization. Given this source code, distributed blocks will be generated for \texttt{proxy()}, \texttt{fetch()}, and each of the compression engines.
Figure 4.19: Skeleton pseudo-code for FlyWheel

```c
void compress_txt(unsigned char data[]) {
    #pragma HLS DISTRIBUTE
    // text compression algorithm
}

void compress_audio(unsigned char data[]) {
    #pragma HLS DISTRIBUTE
    // audio compression algorithm
}

void compress_img(unsigned char data[]) {
    #pragma HLS DISTRIBUTE
    // image compression algorithm
}

void fetch(char url[], WebPage *page) {
    #pragma HLS DISTRIBUTE
    // empty wrapper to generate communication
}

void proxy(char url[], WebPage *compressedPage) {
    WebPage page;

    fetch(url, page);
    for (int i = 0; i < page.numElements; i++) {
        switch (page.elementKind[i]) {
            case IMAGE:
                compress_img(page.element[i]);
                break;
            case TEXT:
                compress_txt(page.element[i]);
                break;
            case AUDIO:
                compress_audio(page.element[i]);
                break;
        }
    }
    memcpy(compressedPage, &page, sizeof(WebPage));
}
```
Chapter 5

Proof of Concept

In Chapter 4, we proposed an augmentation to Vivado HLS that enables it to express master-worker distributed systems, and defined the hardware blocks that should be generated. In this chapter we provide a proof-of-concept of said enhancement. That is, our goal in this chapter is to provide a complete path to the implementation of the DISTRIBUTE pragma, to demonstrate that the design of the distributed blocks allow them to be combined easily to construct distributed systems, and that significant effort savings are possible through the DISTRIBUTE pragma. Here, we describe our prototype implementation of the DISTRIBUTE pragma for Vivado HLS, as well as two applications that are built using this pragma. We present results that speak towards the feasibility of our ideas and potential for real-world application. Our metric of success is that the distributed systems built using the generated distributed blocks function properly in a system, that significant effort savings are made, and that our prototype does not introduce an inordinate amount of overhead.

5.1 Prototype Implementation

We refer to our prototype implementation of the DISTRIBUTE pragma, that we now detail, as HLS of Datacenter Services (HDS). HDS was built with two purposes in mind. First, the process of building a functioning prototype acted as an exploratory exercise. The challenges and algorithms needed to realize this HLS feature only came into focus as we attempted to implement our enhancement. Furthermore, the design of our programming model, parameters of the pragma, and generated distributed blocks are driven by our experiences implementing its support. Second, HDS enables us to substantiate our work with concrete results. As we will discuss in Section 5.2, our experiments show that our implementation works, to some extent, and that systems built using distributed
blocks can trivially scale from a fully serialized implementation to a large-scale deployment.

5.1.1 Tool Inputs and Outputs

HDS is a source-to-source transformation tool built using code analysis and refactoring libraries provided by Clang [37] and LLVM [38]. Our choice to add support for DISTRIBUTE in this manner rather than modifying an HLS tool has two motivating factors. First, we wish to keep development cycles focused on demonstrating the basic ideas described in this thesis. Augmenting an HLS tool requires significantly more effort than creating a source-to-source utility. Second, the Vivado suite [39] – a mature, production grade, HLS tool and synthesis flow – was readily available to us. This presented the unique opportunity to build atop the state-of-the-art and enabled us to create bitstreams from the generated distributed blocks.

The input to HDS is a single-threaded application written in Vivado HLS C, annotated with DISTRIBUTE pragmas. In our programming model, presented in Chapter 4, the user marks sub-functions with the DISTRIBUTE pragma to turn them into discrete worker engines. Automatic parallelization, an optional feature of the DISTRIBUTE pragma that partitions arrays belonging to the distributed function for simultaneous execution, may be enabled by supplying an elements constant and a partition size. The code may also contain pragmas that are natively supported by Vivado HLS, such as PIPELINE, DATAFLOW, and INTERFACE. Source code is passed to HDS by specifying the working directory of a Vivado HLS project. HDS uses the Vivado HLS generated ‘script.tcl’ to identify all relevant source files as well as the project’s top-level function.

HDS outputs Vivado HLS C that implements each of the distributed blocks using existing pragmas and features of Vivado HLS C. Distributed blocks have a standard interface, described in Section 4.2.2, that allow them to be easily connected to each other to build large distributed systems. This interface is comprised of an input/output pair of AXI stream FIFOs used for packetized communication, an ID port used to configure the block with a GUID, and a master AXI port for each persistent storage (i.e. global variable) that the distributed function accesses. To create IP blocks that are recognized by Vivado IP Integrator, the user must manually invoke Vivado HLS on each distributed block’s top-level function. Although invocation of Vivado HLS on each distributed block is not difficult to automate, it is left for future work. For each distributed function,
HDS creates an aggregator/worker function pair, and modifies the distributed function as necessary in an attempt to maintain functional equivalence.

Aggregator functions are master-side injected logic that issue tasks and aggregate responses. The general structure of an aggregator, shown in Figure 5.1, has two for loops that handle issuing tasks and accepting responses, respectively. The top loop partitions work and issues tasks on the outbound AXI stream FIFO, represented by outFifo in the code. Tasks are formed by transmitting the task index (tskIdx), the master’s ID (id), the number of elements processed in the task (elementsProcessed), and any parameters that are read by the distributed function. For array function parameters that are partitioned (i.e., uses the elements variable in its size expression), the Task Issue logic partitions the array by transmitting a non-overlapping portion of the array of size elements variable instead of the array in its entirety. In compliance with the AXI stream protocol, tasks are terminated by transmitting some data, in this case a 0, with the packet boundary signal, TLAST, asserted. The Response Accept logic, complementing Task Issue, accepts responses from workers on the inbound AXI stream FIFO, represented by inFifo, and combines them to form complete results. Aggregation of partial results is done through aggregation functions that HDS determines and generates for each output of the distributed function. An aggregation function is the operation that is performed to combine partial results to form a complete one. In the example shown in Figure 5.1, partial results for param0 are summed together; the aggregation function for param0 is summation.

Worker functions are the worker-side counterpart of aggregator functions – they are HDS injected logic that accept requests and issue responses. The body of a worker function follows the template shown in Figure 5.2. The Task Accept logic receives tasks from masters on the worker’s inbound AXI stream FIFO, inFifo, and unpackages the input parameters. The distributed function is then invoked with this data, and the result is transmitted back to the originating master in the Response Issue logic using the worker’s outbound AXI stream FIFO, outFifo. Responses are comprised of the task index (tskIdx), the number of elements processed (ele), and any function parameters that are output by the distributed function. Like tasks, responses are terminated with a 0 pad that asserts the TLAST signal. Figure 5.3 summarizes the communication between master and worker.
```c
void dist_func_agg(...) {
    int tskRcv = 0, tskIss = 0;
    int offset, dummy, i, j;

    // Task Issue logic
    for (i = 0; i < eleVar / partSize; i++) {
        send(tskIdx, 0, false, outFifo);
        send(id, 0, false, outFifo);
        send(elementsProcessed, 0, false, outFifo);
        send(param0, 0, false, outFifo);
        for (int idx0 = 0; idx0 < 10; idx0++) {
            send(param1[idx0], 0, false, outFifo);
        }
        // other parameters that are read from
        send(0, 0, true, outFifo);
        tasksIssued++;
    }

    // Response Accept logic
    for (tskRcv = 0; tskRecv < tskIss; tskRscev++) {
        recv(tskIdx, inFifo);
        recv(elementsProcessed, inFifo);

        recv(param0Temp, inFifo);
        param0 += param0Temp;

        offset = taskIdx * 10;
        for (j = 0; j < elementsProcessed; j++) {
            recv(param1[offset + j], inFifo);
        }
        // other parameters that are written to
        recv(dummy, inFifo);
    }
}
```

Figure 5.1: Template for aggregation functions, adjusted for readability.
void dist_func_worker(...) {
    int dummy, param0, idx0;
    int param1[];

    while(1) {
        // Task Accept logic
        recv(tskIdx, inFifo);
        recv(aggId, inFifo);
        recv(ele, inFifo);
        recv(param0, inFifo);
        for (idx0 = 0; idx0 < 0; idx0++) {
            recv(param1[idx0], inFifo);
        }
        recv(dummy, inFifo);

        // invoke the distributed function
        int offset = tskIdx * partSize;
        dist_func(..., offset);

        // Response Issue logic
        send(tskIdx, aggId, false, outFifo);
        send(ele, aggId, false, outFifo);
        send(param2, aggId, false, outFifo);
        for (j = 0; j < ele; j++) {
            send(param3[j], aggId, false, outFifo);
        }
        send(0, aggId, true, outFifo);
    }
}

Figure 5.2: Template for worker functions, adjusted for readability.

Figure 5.3: Figure depicting communication protocol between masters and workers.
The data transmission code that HDS generates is written in terms of a communications library that we provide. This library is comprised of a set of overloaded \texttt{send()} and \texttt{receive()} functions that handle marshalling and transmission of data to a specified destination. There is a \texttt{send()} and \texttt{receive()} function pair for each supported data type. The excerpt in Figure 5.4 shows the implementation for transmission and reception of a character, an 8-bit value.

The \texttt{send()} function takes as input the character, the ID of the destination which may be a worker kind ID, a Boolean describing whether the data is the last in the packet, and a reference to the FIFO that the data will be transmitted on. Send functions marshal data using a union structure. The data is loaded into the union and its bits literally reinterpreted into a 32-bit unsigned value for transmission. The data is packaged into a packet structure that instructs Vivado HLS how to drive the AXI stream signals.

Receive functions do the opposite of \texttt{send()} functions – they unpack data and return them to the caller. The \texttt{recv()} function takes as input a reference to the location that the data should be returned in and a reference to the FIFO that data is being received on. To de-marshal the data, it loads the raw packet data into a union structure and reinterprets the bits as the desired type. In the snippet shown in Figure 5.4, the code is converted into a character and returned to \texttt{recv()}’s caller through a reference function parameter.

Finally, the code that is output by HDS also has the distributed functions parallelized and adjusted for functional equivalency, if automatic partitioning is enabled. For our implementation, these algorithms are heuristics, and do not correctly parallelize or maintain functional equivalency for all possible input sources. At this time, the modifications that HDS makes are correct only for a very limited set of applications, sufficient to demonstrate a proof-of-concept of our proposed HLS programming model enhancement and usage of the distributed blocks that are generated. An interesting future direction is to determine a more generalized approaching, borrowing algorithms from prior automatic parallelization endeavors such as DMCs that were detailed in Section 3.2.

5.1.2 Tool Flow

HDS is structured as a chain of smaller tools, called \textit{sub-tools}, built using code analysis and refactoring infrastructure provided by Clang [37] version 3.5.0 and LLVM [38] version 3.8.0. The tool flow, shown in Figure 5.5, has three phases: parsing, analysis, and
typedef struct {
    uint32_t data;
    uint32_t dest;
    bool last;
} packet_t;

typedef union {
    uint32_t ui;
    int32_t i;
    float f;
    bool b;
    char c;
} convert_t;

void send(char c, uint32_t dest, bool last, stream<packet_t> & outFifo) {
    packet_t packet;
    convert_t convert;

    convert.c = c;
    packet.data = convert.ui;
    packet.dest = dest;
    packet.last = last;
    out.write(packet);
}

bool recv(char & c, stream<packet_t> & inFifo) {
    packet_t packet = in.read();
    convert_t convert;
    convert.ui = packet.data;
    c = convert.c;
    return packet.last;
}

Figure 5.4: Declaration of the packet_t structure and call signatures for the send() and recv() functions for transmitting a character (signed 8-bit value). Adjustments were made for readability.
Figure 5.5: The parts of our source-to-source transformation tool and how they interact with each other.

<table>
<thead>
<tr>
<th>C Construct</th>
<th>Information Collected</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISTRIBUTE pragma</td>
<td>source location, pragma parameters, full-text of pragma, name of function where it resides</td>
</tr>
<tr>
<td>INTERFACE pragma</td>
<td>source location, full-text of pragma, name of function where it resides</td>
</tr>
<tr>
<td>Constant Variables named by DISTRIBUTE pragma</td>
<td>name, value</td>
</tr>
<tr>
<td>Functions</td>
<td>name of the function</td>
</tr>
<tr>
<td>Arrays</td>
<td>name of the array</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of C constructs that are supported by HDS and the information that Pragma Parser collects in regards to each of them.

rewriting. A Python script is used invoke each sub-tool in succession and fork a copy of all input sources prior to transformation.

The first stage of HDS is the parsing phase, which plays the role of lowering the input Vivado HLS C to LLVM intermediate representation and providing downstream subtools with relevant information regarding the original source. It is comprised of Clang and the Pragma Parser, which are respectively responsible for these tasks. As detailed in Table 5.1, Pragma Parser collects information about INTERFACE and DISTRIBUTE pragmas, constant variables named as elements variables by DISTRIBUTE pragmas, and provides a list of all functions and arrays that exist in the input program.

The Distribute Mutator is an LLVM analysis pass that determines what changes to the input source are needed to distribute it. Given the LLVM IR and DISTRIBUTE pragma information, Distribute Mutator generates a list of instructions on how to modify the source code, called *modification requests*. Modification requests are code mutation operations that are supported by the Rewriter. For instance, the Callsite modification request
Chapter 5. Proof of Concept

<table>
<thead>
<tr>
<th>Modification Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregator Generation</td>
<td>Generate an aggregation function (i.e. suffix _agg)</td>
</tr>
<tr>
<td>Replace Constants</td>
<td>Replaces all uses of a constant variable with a value</td>
</tr>
<tr>
<td>Worker Generation</td>
<td>Generate a worker function (i.e. suffix _worker)</td>
</tr>
<tr>
<td>Add Parameter</td>
<td>Add a parameter to a given function declaration</td>
</tr>
<tr>
<td>Add Pragma</td>
<td>Add a pragma to a function</td>
</tr>
<tr>
<td>Alter Subscript</td>
<td>Alter the subscript function of an array access</td>
</tr>
<tr>
<td>Add Header</td>
<td>Include an additional header in a target file</td>
</tr>
<tr>
<td>Callsite</td>
<td>Re-route all invocations of a function to another one</td>
</tr>
<tr>
<td>Delete Pragma</td>
<td>Delete a pragma from a function</td>
</tr>
<tr>
<td>Set Array Size</td>
<td>Adjust the size of an array declaration</td>
</tr>
<tr>
<td>Alter For</td>
<td>Adjust the parameters of a for loop</td>
</tr>
</tbody>
</table>

Table 5.2: Summary of supported modification requests.

re-routes all invocations of a function to another. The Worker Generation modification request, as another example, generates a worker function for a distributed function, given lists of function parameters that the distributed function reads and writes. Table 5.2 gives a brief description of all supported modification requests.

The Rewriter, built using Clang code refactoring libraries, is responsible for executing modification requests. For each modification request that it receives, it registers a callback that is invoked for each match of the constructs that the modification operates on. For example, a callsite modification callback is invoked for each instance an invocation to the function is found. The rewriter blindly applies changes – it contains no logic that verifies syntax or functional equivalency.

5.1.3 Transformation Algorithms

At the heart of HDS are the transformation algorithms that reside in the Distribute Mutator. Figure 5.6 shows the different algorithms that are run in Distribute Mutator and their interactions.

The first two stages determine an input source’s fitness to be transformed into a master-worker distributed model. Error Checking verifies that the code is syntactically correct and free of nonsensical pragma parameters. For example, partition sizes that are larger than the arrays that they target have no meaning. Validity Checking, on the other hand, determines whether the transformation algorithms that follow will be able to correctly
mutate the source. Although both steps are important, they are omitted from our implementation as we focus on demonstrating basic functionality and assume the input is correct.

Statefulness Checking is an analysis that determines whether a distributed function accesses a global variable. This is done by examining the data dependency graph at the global variable declaration. The data dependency graph is a directed graph where vertices are instructions and edges point in the direction of flow dependencies. Figure 5.7 shows a portion of an example data dependency graph centered at the declaration of a global variable. To determine if a function accesses a global variable, we check if any of the instructions adjacent to the global variable declaration belong to the distributed function. Statefulness is used by the Function Adjustments and Worker Kind ID Assignment steps.

Parameter Access Checking ascertains whether a function parameter is read, written, or both in the body of its function. To determine if a parameter is read from, we traverse the data dependency graph using a breadth first search (BFS) looking for a load instruction that reads from the parameter’s address. The BFS is rooted at the parameter declaration and expands until all instructions in the function have been visited, or a load or function call instruction is discovered. In this case, the check returns true to denote that the argument is read by the distributed function. We do not recursively explore functions that are invoked by the distributed function as a simplification. The algorithm to check for write access is identical, except that it searches for writes rather than reads.
Lists of read and written parameters are used to build Aggregator Generation and Worker Generation modification requests, which require this information. Only arguments that are read are transmitted from master to worker, and only arguments that are written are transmitted in the opposite direction.

Aggregation Type Detection determines how partial results for each of a distributed function’s outputs should be aggregated. For example, the code in Figure 5.8 that sums the elements of an array can be parallelized by computing partial sums on partitions of the array. The partial sums should then be summed to form a complete answer. We refer to summation in this example as the aggregation function for \texttt{summation()}’s return value and the process of coming to this conclusion as aggregation type detection. We omit aggregation type detection from our implementation. However, prior art has explored this problem [40]. As a temporary solution, HDS assumes that all scalar outputs should be summed and that all arrays should be unioned together.

The purpose of Function Adjustments is to “shrink” the distributed function to operate on a subset of the arrays whose sizes are adjusted by the elements variable, which we refer to as partitioned arrays. HDS applies three rudimentary alterations to distributed functions as heuristics to parallelize the distributed function and maintain functional equivalence. First, all instances of the named elements variable are substituted with
Chapter 5. Proof of Concept

```c
int summation(int n, int array_of_int [SIZE]) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += array_of_int[i];
    }
    return sum;
}
```

Figure 5.8: Code snippet implementing summation of all elements of an array.

the partition size within each distributed function. This resizes all partitioned arrays so that only enough memory to process a partition is instantiated during synthesis. Second, HDS adjusts for-loops belonging to stateless (i.e., no global variable accesses), distributed functions so that only a portion of the original iteration space is traversed, following the example shown in Figure 5.9. The initialization statement is made to take the greater of the original lower bound and the current task’s offset, given by the expression $taskOffset = taskIdx \times partitionSize$. The control statement takes the lesser of the original upper bound and the task’s upper bound. Note that, with this method, the induction variable takes on a subset of the values it does in the single-threaded, unaltered code. In other words, for loops are adjusted so that their induction variables traverse a non-overlapping subset of their original values. As a consequence of the induction variables and parameters taking on their original values, indices that are used to access partitioned arrays also take on their single-threaded values, called global indices. The final heuristic that HDS applies is to convert global indices to local indices, indices that access the partitioned data, by subtracting the partition offset from all indices used to access partitioned arrays. In our example, code that adds one to all elements of an array, shown in Figure 4.5, the induction variable, $i$, traverses all values within the range $[0, n)$. Had this example been partitioned using the DISTRIBUTE pragma, the partition offset should be subtracted from $i$ in the expression used to access `array_of_int`. The latter two steps are skipped for stateful distributed functions since the data is assumed to be correctly partitioned by the user. It is important to note that these algorithms are only valid for a very small number of applications. Our objective, when building HDS, is to provide enough support to demonstrate the expressiveness of our programming model, as well as how to use the distributed blocks that are generated by our HLS enhancement. We also discover the areas where further work is required to build a more complete tool.
for (i=\text{MAX}(\text{offset}, 0); i<\text{MIN}(100, \text{offset}+\text{partition size}); i++) {
    // for loop body
}

Figure 5.9: Shown is a for-loop definition that HDS has adjusted in an attempt to maintain functional equivalency.

The last transformation algorithm that is invoked is Worker Kind ID Assignment. As the name suggests, it is responsible for assigning worker kind IDs to the distributed functions. HDS assigns worker kind IDs starting from 0. It tracks IDs that have been assigned by incrementing the counter. For instance, a counter value of 5 denotes that worker kind IDs belonging to $[0, 5)$ have been assigned. When a distributed function is stateful, a range of worker kind IDs equivalent in length to the maximum number of partitions is assigned. The maximum number of partitions is determined by dividing the original value of the elements variable by the desired partition size. In the example code shown in Figure 4.5, the named elements variable has an original value of 10000 and the desired partition size is 100. Therefore, there are a maximum of \( \frac{10000}{100} = 100 \) partitions.

5.1.4 Limitations

The primary objective of HDS is to demonstrate a proof-of-concept. We enumerate the limitations of our support and simplifications that were made to keep our efforts focused on this goal.

Data Types

HDS only supports basic data types that fit within 32 bits, and their arrays, such as single-precision floats, Booleans characters, and integers.Aggregate types such as structures and unions are not supported. Pointers and typedefs have also been left from our implementation. By restricting data type bitwidths, we remove the need for (de)serialization logic used to (de)marshal variables into the input/output AXI stream ports used for their transmission. Table 5.3 summarizes the types that are supported in our implementation.

DISTRIBUTE Nesting

DISTRIBUTE nesting is the invocation of a distributed function by another that is partitioned on the same elements variable. For example, the programmer may write code resembling that which is shown in Figure 5.10 intending to build a hierarchy of
Chapter 5. Proof of Concept

<table>
<thead>
<tr>
<th>Supported Types</th>
<th>Unsupported Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit integer</td>
<td>structures</td>
</tr>
<tr>
<td>32-bit floats</td>
<td>unions</td>
</tr>
<tr>
<td>boolean</td>
<td>pointers</td>
</tr>
<tr>
<td>character</td>
<td>64-bit values</td>
</tr>
</tbody>
</table>

Table 5.3: Categorization of supported and unsupported data types. Note that multi-dimensional arrays are supported, but only of the supported data types.

```c
const int SIZE=1000000000; // one billion

void level2(int array[SIZE]) {
   #pragma HLS DISTRIBUTED elements.var=SIZE partition.size=100
   // function body
}

void level1(int array[SIZE]) {
   #pragma HLS DISTRIBUTED elements.var=SIZE partition.size=1000
   level2(array);
}

void top_level(int array[SIZE]) {
   level1(array);
}
```

Figure 5.10: Skeleton code demonstrating DISTRIBUTED nesting.

master-slave relationships, like in Figure 5.11. Doing this shifts some of the burden of partitioning data, issuing requests, and aggregating responses from `top_level()` to its workers – the act of distributing work has been made distributed.

DISTRIBUTED nesting is useful when the data is divided into many partitions. In such a situation, it may be the case that the master bottlenecks execution since it is unable to issue requests or aggregate responses rapidly enough to fully leverage task-parallelism. Production systems such as Dremel, Google’s large scale SQL processing system [4], use this technique. Figure 5.12 shows Dremel’s architecture. SQL queries are gradually divided into sub-queries as they flow from the root node, through intermediate nodes to leaf nodes. As sub-query results propagate back up the hierarchy to the root node, they are gradually aggregated. Although valuable, DISTRIBUTED nesting does not contribute to a proof-of-concept of the DISTRIBUTED pragma. Support for this feature is left for future work.
Figure 5.11: Block diagram showing hierarchical master-worker relationships. The acts of distribution and aggregation have themselves been distributed.

Figure 5.12: Block diagram showing the hierarchical structure of Dremel, Google’s large scale SQL processing system. Taken from [4].
Error Checking

HDS does not tolerate syntactical errors and nonsensical pragma parameters. Since we seek only to provide prototype support, we do not implement any error checking. All input Vivado HLS C, including the DISTRIBUTE pragma, are assumed to be syntactically pristine and reasonable. For example, we do not account for partition sizes that are larger than the maximum size of the array being partitioned. Of course, these algorithms should be implemented if the DISTRIBUTE pragma is to be supported in a production tool.

Validity Checking

Validity checking determines whether input Vivado HLS C, devoid of errors, would be correctly mutated by our transformation algorithms. Formally, validity checking bifurcates the space of all possible input Vivado HLS C programs into those that do and do not maintain functional equivalency when our transformation algorithms are applied. Although these algorithms are important, they are left as future work. As is, HDS indiscriminately applies our transformations to any input source code.

Aggregation Type Detection

Aggregation type detection is the process of determining how to consolidate intermediate results into a complete one. These algorithms are necessary to determine an aggregation function for each output of distributed functions. Our current solution assumes that all scalar outputs should be summed together and that partitions of arrays should be appended to one another.

5.2 Case Studies

We evaluate our prototype support for the DISTRIBUTE pragma by implementing two applications using our Vivado HLS enhancement. In the sections that follow, we describe these applications and present results that speak towards key metrics used to evaluate ease-of-use proposals like our own. The goal here is to show that HDS works – that the distributed blocks it generates can be combined to build distributed systems, and that the overheads that are introduced are not debilitating. In both cases, the applications are synthesized and compiled using Vivado IP Integrator [41], Vivado HLS [5], and SDK [42] version 2016.2. Our testbed a single ZedBoard [43] equipped with the Zynq XC7020-1CLG484 [44] CPU-FPGA hybrid part clocked at 100 MHz. It is important to note
const int MAX_ELE = 1000;

void add_n(int array_of_int [MAX_ELE], int ele, int n) {
    #pragma HLS DISTRIBUTE partition_size=10 elements=MAX_ELE
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < ele; j++) {
            array_of_int[j]++;
        }
    }
}

void top_add_n(int array_of_int [MAX_ELE], int n) {
    add_n(array_of_int, MAX_ELE, n);
}

Figure 5.13: Source code for our add N microbenchmark. Pragma already part of the Vivado HLS 2016.2 [5] specification have been omitted for brevity. Our contribution is only the DISTRIBUTE pragma.

that we only deploy our benchmarks on a single FPGA to simplify our testing. A real deployment should span many FPGAs so that the distributed system constructed from our generated distributed blocks enjoys high throughput and reliability. The difference in scale between our experiments and a full-scale system is the reliability and latency of the interconnects.

5.2.1 Add N

Our first application is that of adding an arbitrary integer, N, to all elements of an array. It is a contrived microbenchmark intended to demonstrate a proof-of-concept under the most ideal conditions. Our implementation, shown in Figure 5.13, is intentionally inefficient. It adds one to every element of the input array, N times, to emulate an application where the compute time dominates overheads due to task issue, load-balancing, and communication. So that there is ample opportunity for simultaneous execution, MAX_ELE is declared as the elements variable and is set to be partitioned into partitions of size 10. By doing so, top_add_n() will generate \(\frac{1000}{10} = 100\) parallel tasks during execution.

Given the source shown in Figure 5.13 as input, our tool generates distributed blocks implementing top_add_level() and add_n(), shrunk to behave as though SIZE=10. Only a single worker kind ID, 0, is assigned to the add_n() worker – any add_n() worker instance
can correctly process any task destined for an add_n() worker. The complete generated source code, adjusted for readability, is available in Appendix A.

To build a functioning distributed system, we integrate several instances of the generated distributed blocks alongside supporting hardware as shown in Figure 5.14. The distributed system is comprised of one top_add_n() instance that submits work to up to fourteen add_n() worker instances through a scheduler. Each of the distributed blocks is arbitrarily assigned a GUID that does not intersect with the add_n() worker kind ID range, \{0\}. The IP blocks that were used in our project are summarized in Table 5.4.

The scheduler is a round-robin load balancer that we provide, and is not generated by HDS. Configured with the IDs of all add_n() worker instances, it replaces the destination field of incoming packets with a worker’s ID and returns the packet to the AXI Stream Interconnect. During operation, the scheduler can be set to route packets to some or all of the add_n() worker instances to adjust the number of workers that contribute to the computational effort. It is likely that, for production systems, a load-balancing algorithm more sophisticated than round-robin will be used. We chose to use round-robin for our experiments for simplicity.

All distributed blocks and the scheduler communicate over an AXI Stream Interconnect, a packet switch provided by Vivado IP Integrator. It is configured to forward packets destined for GUID=0 to the scheduler, and all other packets to the distributed block whose assigned ID matches the destination of the packet. For example, a packet that sets its destination ID=2 will be routed to the port that connects to the block with ID=2, in this case the first add_n() worker instance. The AXI Stream Interconnect is set to operate in store-and-forward mode – cut-through does not seem to function correctly. Each port is given 1024 bytes of buffer space, which is sufficient to hold several packets.

The ZYNQ processing system and accompanying AXI Interconnect bus exist only to facilitate our tests. The ZYNQ processor performs IP core initialization, launches tests, and times execution.

It is possible to build much larger distributed systems using the provided distributed blocks. A production system, for example, may have many top_add_n() instances committing work to many more instances of add_n() workers. Our programming model places no restriction on the number of distributed blocks that may be instantiated. In our small-
Figure 5.14: Block diagram of our Add N microbenchmark. The ZYNQ processing system exists only for configuring the system and timing execution.

<table>
<thead>
<tr>
<th>IP Core Name</th>
<th># of Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>top_add_n</td>
<td>1</td>
</tr>
<tr>
<td>add_n_worker</td>
<td>14</td>
</tr>
<tr>
<td>AXI Stream Interconnect</td>
<td>1</td>
</tr>
<tr>
<td>Round-robin Scheduler</td>
<td>1</td>
</tr>
<tr>
<td>AXI Interconnect</td>
<td>1</td>
</tr>
<tr>
<td>ZYNQ Processing System</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.4: The number of instances of each IP block in our design.

scale test however, we are limited by the AXI Stream Interconnect that supports at most sixteen bi-directional AXI stream connections.

**Performance**

In this section, we present results that demonstrate latency reduction as the number of workers is increased. Execution times are averages taken over one hundred runs of our test. Each test’s input vector consists of a randomly generated array of integers, re-computed each iteration, and N set to 200000. By choosing a large N, the compute time dwarfs all overheads. Running on the ZYNQ processor, the xtime library provided by Xilinx is used to record the number of 100MHz CPU cycles that elapse between the
Figure 5.15: Graph showing the execution times of the Add N microbenchmark as the number of workers are increased. As expected, there are diminishing returns as the number of workers increases.

Overhead

In this section, we quantify the area overheads introduced by HDS when converting Add N into its distributed form. Table 5.5 presents the resource utilization of Add N’s sequential and distributed implementations. Resource utilization was obtained by run-
Table 5.5: Resource utilization of Add N with and without injected communication and aggregation logic. Unused resources have been omitted.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Quantity Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
</tr>
<tr>
<td></td>
<td>Add N</td>
</tr>
<tr>
<td>Slice</td>
<td>78</td>
</tr>
<tr>
<td>LUT</td>
<td>178</td>
</tr>
<tr>
<td>FF</td>
<td>286</td>
</tr>
<tr>
<td>BRAM</td>
<td>2</td>
</tr>
</tbody>
</table>

At first glance, HDS appears to inject significant area overhead – the sum of the areas of the distributed blocks is far greater than the sequential form of Add N. However, there are a couple of mitigating factors. First, the overhead area pales in comparison to our test platform, the XC7020-1CLG484, an entry-level FPGA. LUT usage, for example, increases by a factor of $\frac{299+373}{178} = 3.8$ times after Add N is made distributed. However, this corresponds to only $299+373−178 = 494$ more LUTs used in the distributed system, only $\frac{494}{53200} = 1\%$ of the number of LUTs on our test FPGA. Second, HDS creates buffers for each of the distributed function’s parameters, causing the memory usage of distributed blocks to grow with the amount of data that is communicated between masters and workers (i.e., reside in the parameter list). In this case, a large quantity of LUTRAM is created for `array_of_int`, which is transmitted in its entirety. Third, the simplicity of Add N means that few resources are required to implement it. This exaggerates the percentage area increase. As we will show in Section 5.2.2, a complex application such as web search that has relatively little communication has little area overhead.

**Effort Savings**

To illustrate engineering effort saved, we enumerate the steps that must be taken to transform Add N into its distributed form in the absence of HDS.

The engineer must first establish the overall distribution architecture. This entails determining what data needs to be partitioned and how intermediate results should be partitioned. For Add N, `array_of_int` is cut into $\frac{1000}{10} = 100$ pieces and each task requests
to add N to its portion of the array. When all partitions of $array\_of\_int$ are returned, they are spliced together to form the complete array and the result is returned to the user.

The next step is to establish the flow of data between master and workers. The engineer must carefully analyze Add N and conclude that both N and $array\_of\_int$ are read from, but only $array\_of\_int$ is written to. Therefore, both N and $array\_of\_int$ should be transmitted from the master to the workers, but only $array\_of\_int$ should be sent in the opposite direction.

The communication protocol must also be designed. Add N places two requirements. First, tasks must be indexed so that partitions of $array\_of\_int$ can be correctly reassembled. Second, workers must be able to name a specific master when transmitting a response since multiple masters may share workers. HDS uses the same communication protocol template, which was described in Section 5.1.1, for all distributed applications.

Finally, code must be generated and modified based on the design. Additional logic is needed to communicate the function parameters between master and workers per the communication protocol. The array, $array\_of\_int$, must be portioned prior to transmission and put back together after being updated.

The culmination of these steps is the HDS generated source code that is shown in Appendix A. It is apparent that there are significant differences between it and the original source code that was presented in Figure 5.13. In effect, HDS can transform a single-threaded program into one that can run on a distributed system by doing the data partitioning and packaging the components so that they can properly communicate data.

### 5.2.2 Web Search

Our second benchmark is a simplified version of web search, following the scheme laid out in Section 4.4.1. Our implementation is not intended for direct comparison with production search engines. To compare against such systems, one must implement precisely the same algorithms and produce precisely the same outputs for the same inputs. However, these algorithms are closely guarded secrets and are very complex. Even to implement Google in its original form as proposed in 1998 [45] would be a significant endeavor. The goal here is to demonstrate that our proposed concept can work.
We offer our simplified version of web search as a more meaningful experiment than a microbenchmark, one that is more representative of the intended use case. In our implementation, we assume that all words are exactly three letters, and that the space of all websites is limited to three-character URLs ranging from “www.aaa.com” to “www.wex.com”. This represents a reverse index that maps $26^3 = 17576$ possible 3-letter words to 15000 websites, the largest possible reverse index that our development tools can compile without running out of heap space. Because of these simplifications, we can treat each query word as a radix 26 number. The hashing function used to look up matching document IDs in the reverse index, labelled $h(x)$ in Figure 4.15, merely converts each three-letter word into its decimal equivalent. To maintain precision in our explanation, we limit the remainder of our description to the most complex case that we profile, where the reverse index, doc ID, and URL table are cut into quarters. However, the same design principles apply for any partitioning of these data structures.

Given the source shown in Appendix B, our tool produces distributed blocks corresponding to `webSearch()`, `idxServer()`, and `docServer()`, adjusted to operate on a quarter of the reverse index, doc ID table, or URL table. Because web search is a stateful application, our tool assigns the worker kind ID range $[0, 3]$ to `idxServer()` and $[4, 7]$ to `docServer()`. Recall that the length of the worker kind ID range is equivalent to the number of partitions, which is four for our discussion.

One caveat of our implementation is that the aggregation of the document IDs and matching URLs needed to be corrected by hand as HDS did not generate the correct aggregation function. HDS currently assumes that partitions of a divided array should be appended together, as shown in Figure 5.16. The assumption of this aggregation technique is that at most one partition will have partially valid and partially invalid data, and that all following partitions contain purely invalid data. If instead, invalid data is spread across multiple partitions as shown in Figure 5.17, the valid data must be combined to form a contiguous block of valid data. This caveat is a direct result of not supporting aggregation type detection, which was discussed in Section 5.1.4.

To understand our integrated system, it is helpful to have a view of our design at an architectural level, shown in Figure 5.18. The core distributed system is coloured red. It is comprised of a single web search master and four instances each of Index Servers and Document Servers. Recall from Chapter 4 that there must be at least one worker instance created for each worker kind ID assigned to a distributed function. In compliance, we
Chapter 5. Proof of Concept

Figure 5.16: Diagram showing correct aggregation of an array that has been linearly partitioned. Grey represents valid data, whereas white represents unused memory or nonsensical data.

Figure 5.17: Diagram showing correct aggregation of an array where each partition has invalid data. Grey represents valid data, and white represents unused memory or nonsensical data.
Figure 5.18: Architectural view of our experimental web search. The core distributed system has been coloured red while paths associated with off-chip DDR have been coloured blue.

Instantiate exactly one Index Server for each of the four worker kind IDs it is assigned. For the same reasons, we instantiate four Document Servers. Each index and Document Server instance is connected to a private DDR RAM, coloured blue.

Figure 5.19 gives our block diagram, colour coded to match our architectural diagram. Its constituents are summarized in Table 5.6. A challenge that arises as a result of using a single ZedBoard is that only one DDR controller, resident in the ZYNQ processor, exists. To give the illusion of a dedicated RAM, we partition and assign a single contiguous block of memory to each Index and Document Server. Figure 5.20 shows the layout of the DDR RAM address space. RAM partitions are assigned by hardwiring the offset port of the index and document server blocks to the base address of the RAM block it is assigned. Like in Section 5.2.1, the ZYNQ processing system is used for profiling and the AXI Stream Interconnect provides the connectivity between all distributed blocks. However, packets here are directly forwarded to a distributed block with no intermediate load-balancer. Since each worker kind ID is assigned a single distributed block, all traffic directed at a worker kind ID can be directly forwarded to that IP core.
Figure 5.19: Block diagram of our web search implementation with 4 partitions of the reverse index and URLs table. The paths corresponding to DDR connections are shown in blue.

<table>
<thead>
<tr>
<th>IP Core Name</th>
<th># of Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>web_search</td>
<td>1</td>
</tr>
<tr>
<td>doc_server</td>
<td>1 - 4</td>
</tr>
<tr>
<td>idx_server</td>
<td>1 - 4</td>
</tr>
<tr>
<td>AXI Stream Interconnect</td>
<td>1</td>
</tr>
<tr>
<td>AXI Interconnect</td>
<td>1</td>
</tr>
<tr>
<td>ZYNQ Processing System</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.6: The number of instances of each IP block in our design.
Chapter 5. Proof of Concept

Figure 5.20: layout of DDR memory.

Performance

In this section, we present query execution latency measurements of our implementation of web search as the number of Reverse Index Servers and Document Servers is gradually doubled to four instances. Each datapoint is an average taken over four-hundred trials, where each test iteration records the time taken to service a randomly generated query. Search queries are three words long, and each word is a randomly generated sequence of three characters. One possible search query, for example, is “dfy, qox, wlo”.

The Reverse Index and URLs Table are randomly generated a priori, and collectively map $26^3 = 17576$ possible 3-letter query words to 15000 possible web pages. Each web page, represented by a document ID, is set to have a probability of $\frac{1}{20}$ to match with a query word, which corresponds to a $\frac{1}{8000}$ chance to match with a three-word query. It follows that, for our Reverse Index that maps 17567 words to 15000 web pages, 1.875 web search results are expected for any query word.

\[
P(\text{wordMatch}) = \frac{1}{20} \quad (5.1)
\]

\[
P(\text{queryMatch}) = P(\text{wordMatch} \cap \text{wordMatch} \cap \text{wordMatch}) = \frac{1}{20} \times \frac{1}{20} \times \frac{1}{20} = \frac{1}{8000} \quad (5.2)
\]

\[
E(\text{queryMatches}) = P(\text{queryMatch}) \times 15000 = 1.875 \quad (5.3)
\]

We choose to average our results over four-hundred test iterations so that the average number of search results closely matches the expected. Doing so makes comparison of latency results fair, since execution latency is sensitive to the number of search results. Fewer search results leads to fewer lookups to convert document IDs into URLs during URL Translation. Table 5.7 gives the average number of search results that each of our web search implementations generates. The expected and empirical data closely match one another, demonstrating that our comparison is fair.
# Table 5.7: Percent error between the average number of search results at each datapoint and the expected number of results.

<table>
<thead>
<tr>
<th># of workers</th>
<th>Average Matches</th>
<th>% error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.8625</td>
<td>0.67%</td>
</tr>
<tr>
<td>2</td>
<td>1.885</td>
<td>0.53%</td>
</tr>
<tr>
<td>4</td>
<td>1.8525</td>
<td>1.2%</td>
</tr>
</tbody>
</table>

Trials are timed using the xtine library provided by Xilinx that counts the number of 100MHz ZYNQ cycles between when the top-level is launched until its done signal is asserted. A tight loop is used to poll for this event.

Ideal latency scaling for web search has a lower bound of \( \Omega(n) = \frac{1}{n^2} \) – query latency can at best decrease by a factor of the inverse square of the number of workers. This bound is driven by the document ID intersection logic that uses a doubly-nested loop to ascertain whether each document ID of a running set intersection exists in a new word match set. In the worst case, both bounds scale with the number of document IDs in the reverse index shard, since intersecting the first word match with the second has the most document IDs to compare. As the number of reverse index shards is increased, the number of document IDs in each shard is inverse proportionally decreased. Therefore, the doubly-nested for loop decreases in number of iterations inverse quadratically.

Figure 5.21 graphs the execution latency as the number of reverse index and document workers is increased simultaneously. The blue line represents empirical data, whereas the red line shows ideal latency scaling given by the equation \( f(x) = \frac{0.01739}{x^2} \), normalized to the execution latency when only a single Index Server and Document Server instance exist, 0.01739 seconds. As the number of workers increases, the curves gradually diverge. Table 5.8 lists precisely how much the experimental data deviates from the ideal. When there are four Reverse Index Servers and four Document Servers, the latency misses the ideal by 27.8%, a significant amount. There are several contributing factors that lead to poor scaling in this scenario. First, all distributed blocks access the same DDR memory, creating contention for the DDR controller. Second, the reverse index that is used is significantly smaller than production ones. Because of this, fewer document IDs reside in each shard, causing intersection to be less compute intensive than it normally is.
Figure 5.21: Execution time of web search as the number of partitions is increased.

Table 5.8: Percentage difference between ideal and experimental query execution latencies.

<table>
<thead>
<tr>
<th># of Workers</th>
<th>Experimental Latency [s]</th>
<th>Ideal Latency [s]</th>
<th>∆ [s]</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.017</td>
<td>0.017</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>0.005</td>
<td>0.004</td>
<td>0.0002625</td>
<td>6.03%</td>
</tr>
<tr>
<td>4</td>
<td>0.001</td>
<td>0.001</td>
<td>0.0003031</td>
<td>27.8%</td>
</tr>
</tbody>
</table>

Overhead

Table 5.9 summarizes the resource utilization of web search. Like in Section 5.5, these results are obtained through Vivado HLS by running Export RTL with the evaluate option enabled. All parameters belonging to the sequential implementations of \( \text{idxServer}() \) and \( \text{docServer}() \) were set to be configured over AXILite by labelling them with the INTERFACE pragma.

Our numbers show meager increases in resource utilization when distributing web search. Index Server, for instance, shows a \( \frac{1317 - 1297}{1297} \times 100\% = 1.54\% \) increase in LUT utilization and \( \frac{2151 - 2113}{2113} \times 100\% = 1.8\% \) increase in flip-flop usage. It is interesting to note that the incurred area overheads are larger than those in Figure 5.5 from Section 5.2.1. LUT overhead here, for instance, increased by \( 1825 + 1317 + 744 - (1297 + 701) = 888 \), whereas Add N used \( 299 + 373 - 178 = 494 \) LUTs. Percentage LUT utilization is significantly lower for Web Search than Add N because Web Search is a significantly more complex, and therefore LUT intensive, application than Add N. This result shows that, HDS injects minimal overhead for sufficiently intricate applications.
<table>
<thead>
<tr>
<th>Resource</th>
<th>Sequential</th>
<th>Quantity Used</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Seq Tested</td>
<td></td>
<td>Top Tested</td>
</tr>
<tr>
<td>Slice</td>
<td>562</td>
<td>310</td>
<td>603</td>
</tr>
<tr>
<td>LUT</td>
<td>1297</td>
<td>825</td>
<td>1317</td>
</tr>
<tr>
<td>FF</td>
<td>2113</td>
<td>1031</td>
<td>2151</td>
</tr>
<tr>
<td>DSP</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>BRAM</td>
<td>16</td>
<td>22</td>
<td>5</td>
</tr>
<tr>
<td>SRL</td>
<td>66</td>
<td>0</td>
<td>66</td>
</tr>
</tbody>
</table>

Table 5.9: Resource utilization of web search with and without injected communication and aggregation logic.

**Effort Savings**

We review the code transformations that HDS saves the engineer from doing to demonstrate effort savings. There are two distributed functions in our implementation of web search: Index Server and Document Server.

As with Add N, the first step in the transformation process is to determine how execution and data are to be distributed at a high-level. The GWS should broadcast the search query to each index server cluster. After each cluster has responded with a set of matching document IDs, the GWS should set union the results and transmit the document IDs to each of the document server clusters. The document servers translate the document IDs into URLs, and the result is returned to the end-user by GWS. The number of URLs Table shards and Reverse Index shards is pre-determined and the shards themselves are pre-computed.

The flow of data between GWS and the index servers and document servers must be determined next. By analyzing accesses, HDS determines that the search query and the number of words in the search query should be transmitted from GWS to the index servers. It also determines that the document IDs and number of document IDs should be transmitted back from the index servers to GWS. For the document servers, HDS determines that the document IDs and the number of document IDs should be transmitted from GWS to the document server and that the resulting URLs and the number of resulting URLs should flow in the opposite direction.

Like in Section 5.2.1, the engineer must also design a communication protocol. In addition to the index and document servers being able to address the GWS, the GWS
must be able to name a cluster of index servers or document servers to transmit a task. The GWS transmits one task for each shard of the Reverse Index and URLs Table so that a complete result can be obtained. HDS assigns a worker kind ID range whose cardinality equals the number of partitions. This allows GWS to name a cluster for a particular shard. For example, worker kind ID=0 may correspond to the first reverse index shard, while worker kind ID=5 may correspond to the second URLs Table shard.

Finally, code must be generated with the above considerations in mind. HDS creates code that communicates the queries, document IDs, and URLs to their correct destinations according the communication protocol. The master is made to issue one task per worker kind ID. Miscellaneous logic is injected to complete the transformation process. This includes adding ports for the input output/output FIFO ports, and M_AXI ports to connect the memories that store the URLs Table shards and the reverse index shards.

The result of the transformation process is Vivado HLS C implementing each of the distributed blocks. Appendix C shows the code that is automatically generated by HDS. Significant amounts of code have been added to the original, shown in Appendix B, to create its distributed form. There are clearly significant effort savings here.
Chapter 6

Conclusions and Future Work

We leave the reader with our thoughts on the work presented here and interesting directions for future research in this concluding chapter.

6.1 Conclusions

In this thesis, we propose that HLS tools be augmented to be able to express master-worker distributed systems, envisioning a future where entire datacenter services are implemented on FPGAs. Our work is motivated by the rise in popularity of HLS tools and the application of FPGAs to datacenters. In Chapter 4, we introduce DISTRIBUTE, a function-level pragma that labels a function to be implemented as a discrete hardware engine. We defined the interface and behaviour of the hardware blocks that should be generated, and in Section 4.4, showed how two real-world datacenter applications might be expressed using the DISTRIBUTE pragma. In Chapter 5, we detail HDS, our prototype support for the DISTRIBUTE pragma for Vivado HLS implemented as a source-to-source transformation tool. We evaluate our work by creating a microbenchmark, Add N, and a simplified version of Web Search using the DISTRIBUTE pragma, and present results that speak towards the viability of our ideas.

Our results show that the DISTRIBUTE pragma is a viable idea, and that further work should be invested towards maturing this research direction. HDS could correctly create distributed blocks for our Add N microbenchmark. Furthermore, the distributed systems built using the generated distributed blocks showed ideal scaling – as the number of workers was doubled, the execution time was exactly halved. These results prove that the DISTRIBUTE pragma is feasible, since at least for the ideal parallelization application that is Add N, HDS can generate hardware that reduces execution time by the
best possible amount. Web search also shows encouraging results, showing that real life applications can be expressed with our proposed programming model, and that minimal resource overheads are injected by HDS. However, some intervention was required to correct the generated code to function correctly. Also, the execution latency was significantly sub-optimal; our generated web search executed 30% slower than ideal when four of each kind of worker was instantiated. Therefore, although we demonstrate a proof-of-concept in this thesis, we also show that more work is necessary to bring this research to maturity.

6.2 Future Work

We see this research as a starting point for many interesting research ideas. Some stem from the fundamental notion of tailoring HLS to its target applications, while others develop our idea to completion. Figure 6.1 organizes the possible research directions that we see into an infographic.

6.2.1 Tailoring High-Level Synthesis to Datacenter Applications

The root idea of this thesis is the notion of modifying HLS tools to better fit the needs of its current and future uses. Our work is merely one way that HLS can be adapted to better serve its applications. With HLS continuing to grow in popularity and more applications being ported to FPGAs, further research is necessary to discover creative methods of making HLS more expressive of and effective at synthesizing the applications that designers wish to build.
6.2.2 Algorithms

This thesis focuses on making datacenter applications easier to implement on FPGAs using HLS tools. We propose the DISTRIBUTE pragma and provide a working prototype for its support. Our implementation, however, has many shortcomings that should be addressed. Chief of these are the algorithms that validate input applications and transform them into their distributed forms. Code transformation techniques that are more sophisticated than modifying loop bounds should be developed so that a wider variety of input code can be correctly transformed. Aggregation type detection should be implemented as described in [40]. Finally, validity checking rules should be created together with the transformation algorithms to ensure that functional equivalency is maintained.

There are also many ways that the generated blocks may be optimized. One possibility is to search for opportunities where task execution can be pipelined with task reception, rather than serially receiving and executing tasks. For instance, partitions of array of int in the example that was presented in Figure 4.7 can be streamed through workers as they arrive instead of fully receiving each task prior to executing it. Doing this reduces execution latency, improves throughput, and minimizes on-chip memory utilization. Another optimization opportunity is to forgo transmission of meaningless partitions of arrays. Returning to the example in Figure 4.7, if only a portion of array contains valid data (i.e., \( n < \text{SIZE} \)), then partitions containing only invalid data may be ignored completely, keeping spurious traffic to a minimum.

6.2.3 Adjustable AXI Stream Width

Distributed blocks communicate over the AXI stream protocol that is assumed to have a 32-bit data width. While this is sufficient for our prototype, it is likely that variable width data signals are desirable. By calibrating the bit-widths, the distributed blocks can adjust transmission bandwidth to match that of the communication medium. For instance, if all the distributed blocks happen to be implemented on a single FPGA a very wide data signal may be desirable to fully utilize all available connections. A narrower bit width, on the other hand, is useful when wires are in short supply such as if two distributed blocks residing in different FPGAs communicate over package pins.

6.2.4 Maximize FPGA Utilization

Datacenter operators assiduously attempt to maximize utilization of all resources in the datacenter since doing so leads to major cost savings. Fewer hardware resources are
necessary to run the datacenter applications. In turn, procurement costs are reduced and less power is needed to operate the datacenter. If FPGAs are added to datacenters, they too must be used to their maximum. This may be made especially challenging if FPGAs of differing vendor and resource quantities are used in datacenters. There are two possible research directions here. First, HLS tools can be enhanced to expand designs to fill a target FPGA, relieving much of the burden of re-tuning pragma parameters for each FPGA stock keeping unit that exists in the datacenter. Second, an alternative approach would be to collocate distributed blocks onto FPGAs in a way that maximizes the amount of resources used.

6.2.5 Integration with Existing Services

It is often the case that datacenter applications invoke one another. For example, Dremel [4], Map Reduce [17], and Tensor Flow [11] use Google File System [13] as a scratchpad as well as an input source, and a place to record outputs. There is currently no mechanism in our programming model that allows the user to create distributed blocks that interact with other datacenter applications. Further investigation is necessary to fill this void.
Appendix A

Add N Generated Code

Code generated by HDS for Add N. Indentation and spacing have been adjusted for readability.

```c
#include <hls_stream.h>
#include "../../../lib/transmit.h"
#include "../../../lib/common.h"
#include <stdio.h>
#include <stdlib.h>
using namespace hls;

const int MAX_ELE = 1000;

void add_n(int array_of_int [100], int ele, int n, uint32_t offset) {
    int i, j;

    if (ele <= MAX_ELE) {
        for (i = 0; i < n; i++) {
            for (j = MAX(offset, 0); j < MIN(ele, offset + 100); j++) {
                array_of_int[j - offset]++;
            }
        }
    }
}
```

75
void add_n_agg(stream<packet_t>& in_fifo, stream<packet_t>& out_fifo,
    uint32_t id, int32_t array_of_int [1000], int32_t ele, int32_t n) {

    int offset;
    int tasksReceived;
    int j;
    uint32_t temp;
    int numTasksIssued;

    // issue all the tasks
    for (offset = 0, numTasksIssued = 0; offset < 1000; offset += 100,
        numTasksIssued++) {

        send(numTasksIssued, 0, false, out_fifo); // send the taskIdx
        send(id, 0, false, out_fifo); // send the source
        send(100, 0, false, out_fifo);
        for(j = 0; j < 100; j++) {
            send(array_of_int[j + offset], 0, false, out_fifo);
        }
        send(ele, 0, false, out_fifo);
        send(n, 0, false, out_fifo);
        send(0, 0, true, out_fifo);
    }

    // receive all the results from all tasks
    for(tasksReceived = 0; tasksReceived < numTasksIssued;
        tasksReceived++) {

        int32_t taskIdx;
        int32_t elementsProcessed;

        recv(taskIdx, in_fifo);
        recv(elementsProcessed, in_fifo);
        int32_t offset = taskIdx * 100;
        for (j = 0; j < elementsProcessed; j++) {
            recv(array_of_int[offset + j], in_fifo);
        }
    }
void add_n_worker(stream<packet_t>& in_fifo, stream<packet_t>& out_fifo, 
uint32_t id) {

#pragma HLS INTERFACE axis port=in_fifo 
#pragma HLS INTERFACE axis port=out_fifo 
#pragma HLS INTERFACE ap_stable register port=id 
#pragma HLS INTERFACE s_axilite port=return 

int32_t array_of_int [100];
int32_t ele;
int32_t n;
int i;
uint32_t taskIdx;
uint32_t aggregatorId;
uint32_t temp;
uint32_t elements;
#endif __SYNTHESIS__

while(1) {

//read in a task 
recv(taskIdx, in_fifo);
recv(aggregatorId, in_fifo);
recv(elements, in_fifo);
for (i = 0; i < elements; i++) {
    recv(array_of_int[i], in_fifo);
}
recv(ele, in_fifo);
recv(n, in_fifo);
recv(temp, in_fifo);

// invoke the function on the task
add_n(array_of_int, ele, n, taskIdx * 100);

// send the result of the task back
send(taskIdx, aggregatorId, false, out_fifo);
send(elements, aggregatorId, false, out_fifo);
for (i = 0; i < elements; i++) {
    send(array_of_int[i], aggregatorId, false, out_fifo);
}
send(0, aggregatorId, true, out_fifo);
#endif
Appendix B

Web Search Code

Source code for our simplified implementation of Web Search for benchmarking.

```c
#include <stdio.h>
#include <stdlib.h>
#include "web_search.h"
#include <string.h>

// NOTE: Vivado HLS 2016.1 seems to have trouble exporting RTLs
// of HLS projects which contain global variables that are not
// single pointers to a basic data type.
// Each reverse index entry contains 2 values, the first is the
// offset into docIds that the matches begin, the second is
// the number of doc ID matches there are
int * revIdxRam;
int * docIdsRam;
char * urlsRam;

inline int hash (char str[MAX_QUERY_WORD_LEN]) {
    int val2 = (str[2] - 97);
    int val1 = (str[1] - 97) * 26;
    int val0 = (str[0] - 97) * 26 * 26;
    return (val2 + val1 + val0) * REV_IDX_ENTRY_SIZE;
}

inline bool docIdIsElementOf(int docId, int numDocIds,
```
```c
int docIds[MAX_QUERY_WORDS]) {
    int i;
    for (i = 0; i < numDocIds; i++) {
        if (docId == docIds[i]) {
            return true;
        }
    }
    return false;
}

void indexServer(int numQueryWords, int & numDocIds,
                 char query [MAX_QUERY_WORDS][MAX_QUERY_WORD_LEN],
                 int docIds[NUM_URLS]) {

    #pragma HLS DISTRIBUTE partition_size=250 elements=NUM_URLS
    #pragma HLS INTERFACE m_axi port=revIdxRam bundle=revIdxRam
    #pragma HLS INTERFACE m_axi port=docIdsRam bundle=docIdsRam
    #pragma HLS INTERFACE s_axilite port=return

    int i;
    int j;
    int k;
    int l;
    int localNumDocIds = 0;
    int docIdsBuf[NUM_URLS];
    int revIdxStage[REV_IDX_ENTRY_SIZE];
    int docIdsStage[NUM_URLS];

    // loop through every other query word and intersect its
    // matching doc IDs
    for (i = 0; i < numQueryWords; i++) {
        // look up the query word in the reverse index to
        // get the offset/size into the doc IDs RAM
        int revIdx = hash(query[i]);
        memcpy(&revIdxStage, revIdxRam + revIdx,
             &revIdxStage[REV_IDX_ENTRY_SIZE]);
        memcpy(&docIdsStage, docIdsRam + revIdx,
               &docIdsStage[NUM_URLS]);
    }
```
sizeof(int) * REV_IDX_ENTRY_SIZE);

// look up the docIDs entry
int docIdsOffset = revIdxStage[0];
int docIdsSize = revIdxStage[1];
memcpy(&docIdsStage, docIdsRam + docIdsOffset,
sizeof(int) * docIdsSize);

if (i == 0) {
    for (l = 0; l < docIdsSize; l++) {
        docIdsBuf[l] = docIdsStage[l];
    }
    localNumDocIds = docIdsSize;
}
else {
    // intersect the doc IDs
    int newNumDocIds = 0;
    for (j = 0; j < localNumDocIds; j++) {
        bool intersects = docIdIsElementOf(docIdsBuf[j],
                                            docIdsSize, docIdsStage);
        if (i == 0 || intersects) {
            docIdsBuf[newNumDocIds] = docIdsBuf[j];
            newNumDocIds++;
        }
    }
    localNumDocIds = newNumDocIds;
}

// output all the results
numDocIds = localNumDocIds;
for (k = 0; k < localNumDocIds; k++) {
    docIds[k] = docIdsBuf[k];
}
void docServer(int numDocIds, int docIds[NUM_URLS],
              char urls[NUM_URLS][MAX_URL_LEN], int & numResults) {

    #pragma HLS DISTRIBUTE partition_size=250 elements=NUM_URLS
    #pragma HLS INTERFACE m_axi port=urlsRam bundle=urlsRam
    #pragma HLS INTERFACE s_axilite port=return

    int a;
    numResults = 0;

    // for each doc ID, look up the URL, and copy it to the output
    // urls array
    for (a = 0; a < numDocIds; a++) {
        int urlsRamOffset = docIds[a] * MAX_URL_LEN;
        if (urlsRamOffset < (NUM_URLS * MAX_URL_LEN) && urlsRamOffset >= 0) {
            memcpy(urls[numResults], urlsRam + urlsRamOffset,
                   sizeof(char) * MAX_URL_LEN);
            numResults++;
        }
    }
}

void webSearch(int numQueryWords,
               char query[MAX_QUERY_WORDS][MAX_QUERY_WORD_LEN],
               int & numResults, char result[NUM_URLS][MAX_URL_LEN]) {

    #pragma HLS INTERFACE s_axilite port=numQueryWords
    #pragma HLS INTERFACE s_axilite port=query
    #pragma HLS INTERFACE s_axilite port=numResults
    #pragma HLS INTERFACE s_axilite port=result
    #pragma HLS INTERFACE s_axilite port=return

    int docIds[NUM_URLS];
    int numDocIds = 0;
// look up the matching doc IDs by querying the index servers
indexServer(numQueryWords, numDocIds, query, docIds);

// translate the doc IDs to URLs by querying the doc servers
docServer(numDocIds, docIds, result, numResults);
}
Appendix C

Web Search Generated Code

Code generated by HDS for Web Search. Indentation and spacing have been adjusted for readability.

```c
#include <hls_stream.h>
#include <assert.h>
#include "../../../lib/transmit.h"
#include "../../../lib/common.h"
#include "../url.h"
#include "../doc_id.h"
#include "../rev_idx.h"
#include <stdio.h>
#include <stdlib.h>
#include "web_search.h"
#include <string.h>
using namespace hls;

// NOTE: Vivado HLS 2016.1 seems to have trouble exporting RTLs
// of HLS projects which contain global variables that are not
// single pointers to a basic data type.
// Each reverse index entry contains 2 values, the first is the
// offset into docIds that the matches begin, the second is
// the number of doc ID matches there are
int * revIdxRam;
int * docIdsRam;
```
Appendix C. Web Search Generated Code

char * urlsRam;

inline int hash (char str [MAX_QUERY_WORD_LEN]) {
    int val2 = (str[2] - 97);
    int val1 = (str[1] - 97) * 26;
    int val0 = (str[0] - 97) * 26 * 26;
    return (val2 + val1 + val0) * REV_IDX_ENTRY_SIZE;
}

inline bool docIdIsElementOf(int docId, int numDocIds,
        int docIds[MAX_QUERY_WORDS]) {

    int i;
    for (i = 0; i < numDocIds; i++) {
        if (docId == docIds[i]) {
            return true;
        }
    }

    return false;
}

void indexServer(int numQueryWords, int & numDocIds,
        char query [MAX_QUERY_WORDS][MAX_QUERY_WORD_LEN],
        int docIds[250], uint32_t offset) {

    #pragma HLS INLINE recursive
    #pragma HLS INLINE
    #pragma HLS INTERFACE m_axi port=revIdxRam bundle=revIdxRam
    #pragma HLS INTERFACE m_axi port=docIdsRam bundle=docIdsRam
    #pragma HLS INTERFACE s_axilite port=return

    int i;
    int j;
    int k;
    int l;
int localNumDocIds = 0;
int docIdsBuf[250];
int revIdxStage[REV_IDX_ENTRY_SIZE];
int docIdsStage[250];

// loop through every other query word and intersect its matching
// doc IDs
for (i = 0; i < numQueryWords; i++) {
    // look up the query word in the reverse index to get the
    // offset/size into the doc IDs RAM
    int revIdx = hash(query[i]);
    memcpy(&revIdxStage, revIdxRam + revIdx,
           sizeof(int) * REV_IDX_ENTRY_SIZE);

    // look up the docIDs entry
    int docIdsOffset = revIdxStage[0];
    int docIdsSize = revIdxStage[1];
    memcpy(&docIdsStage, docIdsRam + docIdsOffset,
           sizeof(int) * docIdsSize);

    if (i == 0) {
        // original
        // for (l = MAX(offset, 0);
        //      l < MIN(docIdsSize, offset + 250);
        //      l++) {
        //      //
        //      //      docIdsBuf[l - offset] = docIdsStage[l - offset];
        //      // }
        // modified
        for (l = 0; l < docIdsSize; l++) {
            docIdsBuf[l] = docIdsStage[l];
        }
        localNumDocIds = docIdsSize;
    } else {

// intersect the doc IDs
int newNumDocIds = 0;

// original
// for (j = MAX(offset, 0);
// j < MIN(localNumDocIds, offset + 250); j++) {
//
//   bool intersects =
//   docIdIsElementOf(docIdsBuf[j - offset], docIdsSize,
//   docIdsStage);
//   if (i == 0 || intersects) {
//     docIdsBuf[newNumDocIds] = docIdsBuf[j - offset];
//     newNumDocIds++;
//   }
// }
//

// modified
for (j = 0; j < localNumDocIds; j++) {
    bool intersects = docIdIsElementOf(docIdsBuf[j],
    docIdsSize, docIdsStage);
    if (i == 0 || intersects) {
        docIdsBuf[newNumDocIds] = docIdsBuf[j];
        newNumDocIds++;
    }
}

localNumDocIds = newNumDocIds;

// output all the results
numDocIds = localNumDocIds;
for (k = 0; k < localNumDocIds; k++) {
    docIds[k] = docIdsBuf[k];
}

void indexServer_worker(stream<packet_t>& in_fifo,
stream<packet_t>& out_fifo, uint32_t id) {
#pragma HLS INTERFACE axis port=in_fifo
#pragma HLS INTERFACE axis port=out_fifo
#pragma HLS INTERFACE ap_stable register port=id
#pragma HLS INTERFACE m_axi port=revIdxRam bundle=revIdxRam
#pragma HLS INTERFACE m_axi port=docIdsRam bundle=docIdsRam
#pragma HLS INTERFACE s_axilite port=return

int32_t numQueryWords;
char query[MAX_QUERY_WORDS][MAX_QUERY_WORD_LEN];
int32_t numDocIds;
int32_t docIds[250];
uint32_t taskIdx;
uint32_t aggregatorId;
uint32_t temp;
uint32_t elements;

#ifdef __SYNTHESIS__
    while(1) {
#endif
    //read in a task
    recv(taskIdx, in_fifo);
    recv(aggregatorId, in_fifo);
    recv(elements, in_fifo);
    recv(numQueryWords, in_fifo);
    for (int idx_0 = 0; idx_0 < MAX_QUERY_WORDS; idx_0++) {
        for (int idx_1 = 0; idx_1 < MAX_QUERY_WORD_LEN; idx_1++) {
            recv(query[idx_0][idx_1], in_fifo);
        }
    }
    recv(temp, in_fifo);

    // invoke the function on the task
    indexServer(numQueryWords, numDocIds, query, docIds, taskIdx * 250);

    // send the result of the task back
send(taskIdx, aggregatorId, false, out_fifo);
send(elements, aggregatorId, false, out_fifo);
send(numDocIds, aggregatorId, false, out_fifo);
for (int idx_0 = 0; idx_0 < 250; idx_0++) {
    send(docIds[idx_0], aggregatorId, false, out_fifo);
}
send(0, aggregatorId, true, out_fifo);
#ifdef __SYNTHESIS__
    }
#endif
}

void indexServer_agg(stream<packet_t>& in_fifo,
                      stream<packet_t>& out_fifo, uint32_t id,
                      int32_t numQueryWords, int32_t & numDocIds,
                      char query[MAX_QUERY_WORDS][MAX_QUERY_WORD_LEN],
                      int32_t docIds[NUM_URLS]) {

    int tasksReceived;
    uint32_t temp;
    int numTasksIssued = 0;

    // issue all the tasks
    for (int offset = 0; offset < 1000;
         offset += 250, numTasksIssued++) {

        send(numTasksIssued, 0 + numTasksIssued,
             false, out_fifo); // send the taskIdx
        send(id, 0 + numTasksIssued,
             false, out_fifo); // send the source
        send(250, 0 + numTasksIssued, false, out_fifo);
        send(numQueryWords, 0 + numTasksIssued, false, out_fifo);
        for (int idx_0 = 0; idx_0 < MAX_QUERY_WORDS; idx_0++) {
            for (int idx_1 = 0; idx_1 < MAX_QUERY_WORD_LEN; idx_1++) {
                send(query[idx_0][idx_1], 0 + numTasksIssued,
                     false, out_fifo);
            }
        }
    }
}
send(0, 0 + numTasksIssued, true, out_fifo);

#endif

for (int i = 0; i < numTasksIssued; i++) {
    switch(i) {
    case 0:
        revIdxRam = revIdxVal_0;
        docIdsRam = docIdsVal_0;
        break;
    case 1:
        revIdxRam = revIdxVal_1;
        docIdsRam = docIdsVal_1;
        break;
    case 2:
        revIdxRam = revIdxVal_2;
        docIdsRam = docIdsVal_2;
        break;
    case 3:
        revIdxRam = revIdxVal_3;
        docIdsRam = docIdsVal_3;
        break;
    }
    indexServer_worker(out_fifo, in_fifo, 0);
}
assert(out_fifo.empty());
#endif

// receive all the results from all tasks
for(tasksReceived = 0; tasksReceived < numTasksIssued;
    tasksReceived++) {

    int32_t taskId;
    int32_t elementsProcessed;
recv(taskIdx, in_fifo);
recv(elementsProcessed, in_fifo);
int32_t offset = taskIdx * 250;
int32_t numDocIds_temp = 0;
recv(numDocIds_temp, in_fifo);
//numDocIds += numDocIds_temp; // original
for (int j = 0; j < elementsProcessed; j++) {
    //recv(docIds[offset + j], in_fifo); // original
    recv(docIds[numDocIds + j], in_fifo); // changed
}
numDocIds += numDocIds_temp; // changed
recv(temp, in_fifo);
}
}

void docServer(int numDocIds, int docIds[250],
               char urls [250][MAX_URL_LEN], int & numResults, uint32_t offset) {

#pragma HLS INLINE
#pragma HLS INLINE recursive
#pragma HLS INTERFACE m_axi port=urlsRam bundle=urlsRam
#pragma HLS INTERFACE s_axilite port=return

int a;
numResults = 0;

// for each doc ID, look up the URL, and copy it to the output
// urls array
for (a = 0; a < numDocIds; a++) {
    //int urlsRamOffset = docIds[a] * MAX_URL_LEN; // original
    int urlsRamOffset =
        (docIds[a] - offset) * MAX_URL_LEN; // changed
    if (urlsRamOffset < (250 * MAX_URL_LEN) && urlsRamOffset >= 0) {
        memcpy(urls[numResults], urlsRam + urlsRamOffset,
               sizeof(char) * MAX_URL_LEN);
    }
}
void docServer_worker(stream<packet_t>& in_fifo,
    stream<packet_t>& out_fifo, uint32_t id) {
    #pragma HLS INTERFACE axis port=in_fifo
    #pragma HLS INTERFACE axis port= out_fifo
    #pragma HLS INTERFACE ap_stable register port=id
    #pragma HLS INTERFACE m_axi port=urlsRam bundle=urlsRam
    #pragma HLS INTERFACE s_axilite port= return

    int32_t numDocIds;
    int32_t docIds[NUM_URLS];
    int32_t numResults;
    char urls[250][MAX_URL_LEN];
    uint32_t taskIdx;
    uint32_t aggregatorId;
    uint32_t temp;
    uint32_t elements;
    #ifdef __SYNTHESIS__
    while(1) {
    #endif

    // read in a task
    recv(taskIdx, in_fifo);
    recv(aggregatorId, in_fifo);
    recv(elements, in_fifo);
    recv(numDocIds, in_fifo);
    for (int idx_0 = 0; idx_0 < NUM_URLS; idx_0++) {
        recv(docIds[idx_0], in_fifo);
    }
    recv(numResults, in_fifo);
    recv(temp, in_fifo);

    // invoke the function on the task
docServer(numDocIds, docIds, urls, numResults, taskIdx * 250);

// send the result of the task back
send(taskIdx, aggregatorId, false, out_fifo);
send(elements, aggregatorId, false, out_fifo);
for (int idx_0 = 0; idx_0 < 250; idx_0++) {
    for (int idx_1 = 0; idx_1 < MAX_URL_LEN; idx_1++) {
        send(urls[idx_0][idx_1], aggregatorId, false, out_fifo);
    }
}
send(numResults, aggregatorId, false, out_fifo);
send(0, aggregatorId, true, out_fifo);

#endif

void docServer_agg(stream<packet_t>& in_fifo,
                    stream<packet_t>& out_fifo, uint32_t id, int32_t numDocIds,
                    int32_t docIds[NUM_URLS], char urls[NUM_URLS][MAX_URL_LEN],
                    int32_t & numResults) {

    int tasksReceived;
    uint32_t temp;
    int numTasksIssued = 0;

    // issue all the tasks
    for (int offset = 0; offset < 1000; offset += 250, numTasksIssued++) {
        send(numTasksIssued, 4 + numTasksIssued,
             false, out_fifo); // send the taskId
        send(id, 4 + numTasksIssued, false, out_fifo); // send the source
        send(250, 4 + numTasksIssued, false, out_fifo);
        send(numDocIds, 4 + numTasksIssued, false, out_fifo);
        for (int idx_0 = 0; idx_0 < NUM_URLS; idx_0++) {
            send(docIds[idx_0], 4 + numTasksIssued, false, out_fifo);
        }
    }
send(numResults, 4 + numTasksIssued, false, out_fifo);
send(0, 4 + numTasksIssued, true, out_fifo);
}

#ifndef __SYNTHESIS__
for (int i = 0; i < numTasksIssued; i++) {
    switch(i) {
    case 0:
        urlsRam = urlsVal_0;
        break;
    case 1:
        urlsRam = urlsVal_1;
        break;
    case 2:
        urlsRam = urlsVal_2;
        break;
    case 3:
        urlsRam = urlsVal_3;
        break;
    }
    docServer_worker(out_fifo, in_fifo, 0);
}
assert(out_fifo.empty());
#endif

// receive all the results from all tasks
for(tasksReceived = 0; tasksReceived < numTasksIssued;
    tasksReceived++) {
    int32_t taskIdx;
    int32_t elementsProcessed;
    recv(taskIdx, in_fifo);
    recv(elementsProcessed, in_fifo);
    int32_t offset = taskIdx * 250;
    for (int j = 0; j < elementsProcessed; j++) {
        for (int idx_1 = 0; idx_1 < MAX_URL_LEN; idx_1++){

Appendix C. Web Search Generated Code

```c

void webSearch(int numQueryWords,
    char query[MAX_QUERY_WORDS][MAX_QUERY_WORD_LEN],
    int & numResults, char result[NUM_URLS][MAX_URL_LEN],
    stream<packet_t>& in_fifo, stream<packet_t>& out_fifo,
    uint32_t id) {

    #pragma HLS INTERFACE axis port=out_fifo
    #pragma HLS INTERFACE axis port=in_fifo
    #pragma HLS INTERFACE ap_stable register port=id
    #pragma HLS INTERFACE s_axilite port=numQueryWords
    #pragma HLS INTERFACE s_axilite port=query
    #pragma HLS INTERFACE s_axilite port=numResults
    #pragma HLS INTERFACE s_axilite port=result
    #pragma HLS INTERFACE s_axilite port=return

    int docIds[NUM_URLS];
    int numDocIds = 0;

    // look up the matching doc IDs by querying the index servers
    indexServer_agg(in_fifo, out_fifo, id, numQueryWords, numDocIds,
        query, docIds);

    // translate the doc IDs to URLs by querying the doc servers
    docServer_agg(in_fifo, out_fifo, id, numDocIds, docIds,
```

Note: The text continues from where it was cut off, but it is clear that the rest of the code is related to web search functionality and involves interfacing with the `in_fifo` and `out_fifo` streams to perform operations such as receiving messages, aggregating results, and translating doc IDs to URLs.
result, numResults);
}
Bibliography


