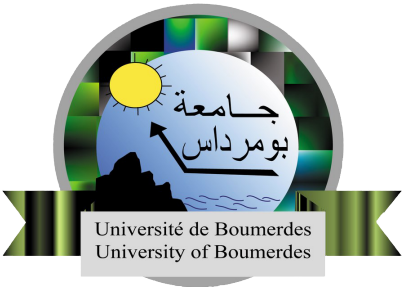
People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University M'Hamed BOUGARA – Boumerdes



Institute of Electrical and Electronic Engineering
Department of Computer Engineering

Final Year Project Report Presented in Partial Fulfilment
of the Requirements for the Degree of

Master

in **Electrical and Electronic Engineering**
Option: Computer Engineering

Hardware/Software Co-Design Framework Implementation for Distributed Computing with FPGA-Based Computing Nodes

Presented by:

- **LEHAMEL Lotfi**
- **KHAROURI Abdenour**

Supervisor:

- **Dr. MAACHE Ahmed**

Registration number: 2021/2022

Hardware/Software Co-Design Framework Implementation for Distributed Computing with FPGA-Based Computing Nodes

Abstract

In the distributed computing field, FPGA integration can offer significant speedups and power efficiency in many applications. The complexity of cooperating hardware and software into a single system remains a limiting factor in clustering FPGAs in a distributed computing environment. In this work, a distributed system that offers users a framework to configure and interface with multiple FPGA boards was designed and implemented. The system consists of a server connected to several Terasic DE10 FPGA boards on a local network, while a custom messaging protocol built on top of TCP/IP is used for communication. The system was built with two factors in mind: flexibility, as in its ability to adapt to various application requirements; and scalability, in which it can support multiple users and multiple computing nodes. In terms of testing, a load-free performance benchmark was conducted on each component of the system separately to observe the system overhead on the user application. Using a matrix multiplier test-case application, the system was tested on localhost to avoid networking hardware limitation. As a result, the system showed a low overhead on the user application while its behavior followed a logical pattern. Incorporating more computing hardware led to an increase in the total throughput reaching up to 8.6 Gbps and a decrease in the latency overhead. To evaluate the achieved results on the test-case application using the system, a comparison with a software implementation of the same application was conducted. The test-case application on the system performed up to 1.3, 5.5, and 11.5 times faster using 4, 16, and 36 nodes, respectively, compared to the software-based implementation.

Acknowledgements

Praise to Allah the Almighty and most merciful for giving us the wisdom, persistence and potency to complete this project. We revere the patronage and moral support extended with love by our families, especially our parents whose financial and constant passionate encouragement made this undertaking a reality.

We express our sincere gratitude to our supervisor, Dr. Maache Ahmed for his guidance and equipment support for the implementation.

Our colleague Mr. Dahmani Abdelmadjid, We would like to give you a special thanks for your great impact and contribution in this work.

Last but not least, Wameedh Scientific Club including its current and old members, none of this would be possible without you providing us with the good company and the suitable working environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Due to the rising demand for processing power across a wide range of application areas, there has recently been a rise in interest in high-performance computing using FPGAs. Over the last two decades, research has shown that hardware acceleration with FPGAs delivers significant performance advantages in some application areas such as Machine Learning, Digital Signal Processing, and Cryptography.

One way of making use of the FPGA's processing power is to group them in a cluster and divide up the computations among them. Distributed computing refers to the idea of breaking a problem down into smaller tasks and distributing these tasks among various processing elements. Performance is increased through parallelization when FPGAs are used in a distributed computing environment.

1.1 Motivation

For many years, Moore's Law[1] and Dennard Scaling[2] have driven continuous improvements in computer technology in terms of both performance and power efficiency. With Moore's Law fading and Dennard Scaling coming to an end, the age of Dark Silicon[3] is closer than ever. In recent years, this has compelled the Distributed Computing community to adopt specialized accelerators. FPGAs are a type of accelerator that has emerged as a more energy-efficient alternative to GPUs. These devices have historically been targeted for low-power and embedded applications. Furthermore, these devices were historically programmed using Hardware Description Languages (HDL), namely Verilog and VHDL, which have a very different programming model than usual software programming languages such as C and Fortran. This has always been a key barrier to FPGA adoption among software developers.

High Level Synthesis (HLS) technologies have been developed to make FPGAs usable by software programmers. Tools enable software developers to define their FPGA design in a common software programming language like C and C++ before converting it to a low-level description based on Verilog or VHDL. Xilinx recently bought AutoESL[4], and based on that, created Vivado HLS[5]. Altera (now Intel Programmable Solution Group introduced their Intel HLS Compiler[6] to provide a similar possibility for software programmers. With official HLS tools being created and maintained directly by FPGA manufacturers, there was a rapid shift in the HLS ecosystem that allowed for more widespread use of FPGAs among software programmers.

This led to exploiting the FPGAs power and efficiency in various applications, namely Machine learning, Digital signal processing and cryptography due to their unique characteristics that allow parallelism to be used.

Machine learning (ML) is a sub field of Artificial Intelligence (AI) and refers to computer systems that have the ability to learn how to solve a problem rather than being explicitly programmed to do so. Artificial Neural Networks (ANNs) are one of the approaches of ML that are known to be used in computer vision and signal processing, leading to the need of a good hardware to manipulate images and signals; And since ANNs are parallel in nature, they can draw greater value from the parallelism that FPGAs possess, resulting to a fast AI inference on the edge. Implementations on FPGAs have also been shown to have significantly lower energy consumption per operation than the equivalent on a GPU, which is a requirement for embedded systems.

Digital signal processing (DSP) is the processing of signals that have been converted to digital form. Speech processing, picture processing, audio processing, information systems, control systems, and instrumentation all use DSP technology. DSP applications have characteristics that expand the amount of parallelism that reconfigurable distributed computing can use. Smaller word widths enable more hardware units to fit on a chip, resulting in greater clock speeds. The Fast Fourier Transform (FFT) is commonly used to compute discrete Fourier transforms (DFTs). An FFT implementation on Splash-2 [7], an FPGA-based array processor, was sped up 23 times over a Sparc-10 workstation[8]. In [9], an FPGA implementation of radix-4 FFT attained speedup factors of 9.4, 10, and 12.5 over a TMS320C5x DSP processor.

Cryptography is the study of encrypting and decrypting data, which is a critical part of attaining computer system security. Cryptographic algorithms are a perfect use case for reconfigurable computing because they are computationally demanding and usually implemented using hardware components like shift registers and permutation networks. Hardware implementation of modular arithmetic, a crucial component of cryptography, is more effective than using fixed-width microprocessor arithmetic logic

units. The RSA algorithm was created by Ron Rivest, Adi Shamir, and Len Adleman as a public key encryption system. The key to RSA is the factoring of big numbers, which is exponentially harder to solve as the size of the numbers grows. There isn't yet a quick and effective factoring algorithm for big numbers. In [10], an FPGA-implemented RSA was about 30 times faster than a 1024-bit software implementation.

1.2 Problem Statement

Unlike general-purpose computing systems, which design hardware and software separately, embedded systems design hardware and software cooperatively. The problem of designing a system that clusters FPGAs in a distributed computing environment necessitates designers' understanding of hardware, software, and networking fundamentals. Using FPGAs to exploit massive amounts of parallelism is no easy task. This is due to three characteristics of FPGA design: long development time, complex hardware/software partitioning, and limited size.

One of the most difficult aspects of FPGA design is the approach that designers must take to a problem. Designing for FPGAs requires employing numerous resources scattered throughout a chip at the same time to achieve significant parallelism. In contrast, software development is often targeted at taking advantage of a microprocessor's ability to execute instructions sequentially. Because the brain is designed to think sequentially, transforming a design into parallel logic takes substantially longer than sequential programming. A high learning curve is associated with learning to utilize FPGA development tools, in addition to the increased time it takes to implement designs in an HDL. These tools are frequently vendor-specific, depending on the FPGA used.

Finding the correct balance between software flexibility and hardware speed while meeting design constraints such as performance, area, designer effort, and so on is a difficulty when developing for FPGAs. It is the designer's responsibility to decide how best to divide an application between a microprocessor component ("software") and one or more specialized co-processors ("hardware"). Partitioning is a particularly hard task since there are sometimes multiple ways to segment a design.

1.3 Objectives

The objectives of the work are the following:

- Supply a framework that provides users with a set of functionalities to configure and use multiple FPGA boards for distributing computations.
- Develop a multi-architecture system capable of distributing computation work across multiple computing nodes.
- Conduct a performance-based analysis on each part of the system using a test-case application and observe the effect of changing both the hardware components and the hardware configuration on the overall performance.
- Compare the performance of the test-case application implemented on the system with the software-based implementation of the application.

1.4 Structure of The Work

Chapter 2 outlines some background concepts needed in this work. While Chapter 3 is about the implementation of the project and the inner working details of it. Next, chapter 4 is the benchmarking of the various parts of the work and discussing and analyzing the results. Finally, chapter 5 is the conclusion and future works based upon this work.

Chapter 2

Background

2.1 FPGAs

A Field Programmable Gate Array (FPGA)[11], is a semiconductor device built around a matrix of configurable logic blocks (CLBs) that allows the development of custom hardware logic for quick prototyping and final system design. The main advantage of FPGAs is their reprogrammability, which allows the user to rapidly program the device based on the changing needs of the larger system into which they are integrated. FPGAs are best suited for the fastest-growing applications of today, such as high performance computing, artificial intelligence (AI) and 5G.

2.1.1 SoC-FPGAs

An SoC-FPGA System on Chip-Field Programmable Gate Array) is the combination of FPGA fabric and a Hard Processor System (HPS)[12] based on processor cores such as ARM Cortex processors. This type of architecture combines the performance and flexibility of FPGAs with the simplicity of microprocessors. For Intel's SoC-FPGAs, ARM Cortex A9[13] is the most commonly used microprocessor.

2.1.2 Advanced eXtensible Interface

The Advanced eXtensible Interface (AXI)[14] is a hardware communication protocol developed by ARM as part of the AMBA standard (Advanced Microcontroller Bus Architecture). AXI has been adopted by Intel and Xilinx as the main communication protocol in SoC-FPGAs.

2.1.3 Embedded Linux

Embedded Linux is a type of operating system (OS) that runs a Linux kernel[15] configured to run on embedded devices with limited resources. The HPS of an SoC-FPGA can be programmed as bare metal (i.e., without an operating system). However, this forces the programmer to implement the entire software stack from scratch in order to implement a simple software application. Running an embedded Linux on the HPS allows you to develop a software application that works on any device that runs Linux.

2.1.4 Intel High Level Synthesis

The creation of RTL (Register Transfer Level) is usually done with HDLs[11] (Hardware Description Languages) such as Verilog and VHDL. They take significantly longer to develop compared to software development, as design and verification are time-consuming processes. High Level Synthesis (HLS) compiles C++ code into an RTL design, which accelerates the development of IP cores (Intellectual Property). For Intel SoC-FPGAs, this translation is done with the Intel HLS Compiler[16], which generates RTL code optimized for Intel FPGAs.

2.1.5 Intel Quartus Platform Designer

Platform Designer (formerly Qsys)[17] is a system integration and easy-to-use GUI tool in Quartus Prime Software. It is used to quickly combine multiple IP cores together, including the HPS. It also generates the interconnect logic required.

2.1.6 DE10-Standard Development Board

The DE10-Standard[18] is a development board made by Terasic combines the Cyclone V (5CSXFC6D6F31C6) SOC-FPGA with several components such as DDR3 memory (connected to the HPS), peripherals such as the ADC (Analog to Digital Converter), I/O ports such as Ethernet, USB, VGA, and on-board USB Blaster II. As shown Figure 2.1, the most important features of the board for this project are:

- 1GB DDR3 Memory.
- 1 Gigabit Ethernet PHY with RJ45 connector.
- Micro SD card socket (Contains the embedded Linux image).
- 5,761 Kb embedded memory (Block Ram).

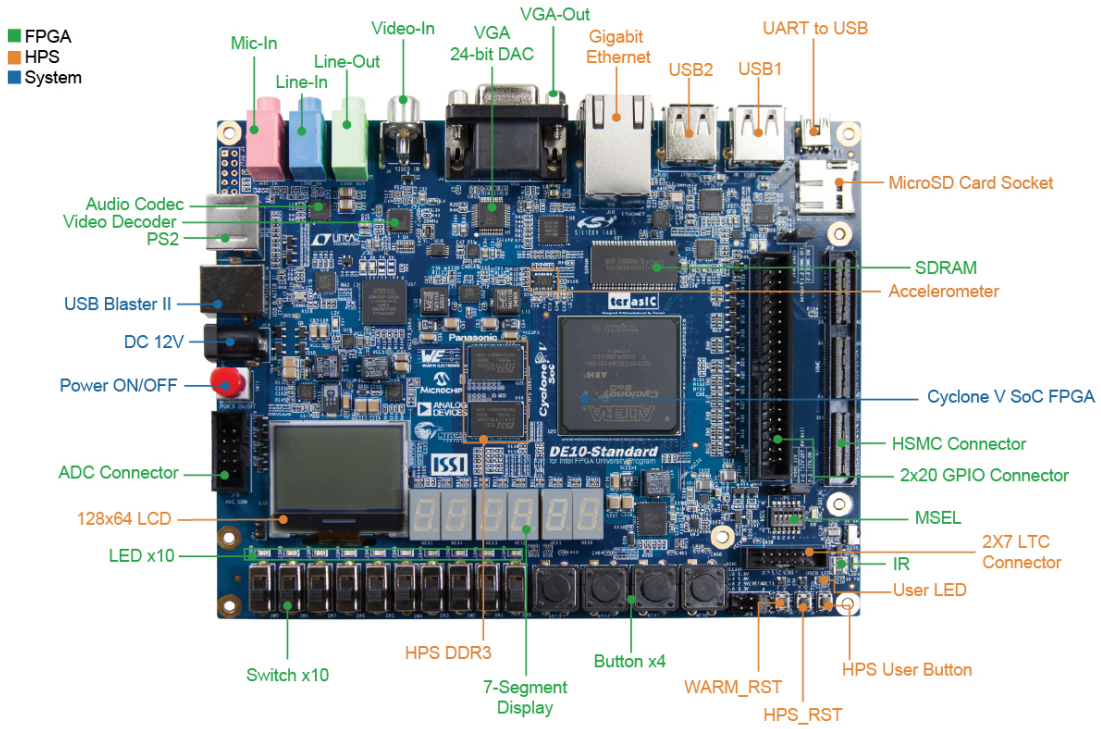


FIGURE 2.1: DE10-Standard Development Board Layout
[19]

2.1.7 Cyclone V

The Intel Cyclone V[20] family is a low-cost and power efficient SoC-FPGA device series that offers a dual-core ARM Cortex-A9 MPCore processor surrounded by a rich set of peripherals and a hardened memory controller. The Programmable Logic side (FPGA fabric), with up to 110K LEs (logic elements), is connected to the hard processor system (HPS) through a high-speed (greater than 100 Gbps) interconnect backbone.

2.1.7.1 Programmable Logic Side

A Field Programmable Gate Array (FPGA) is an integrated circuit consisting of an array of programmable logic cells. The basic architecture of an FPGA consists of a few fundamental elements, such as ALMs, M10K block memory, interconnects, and Input/Output Blocks (IOB).

An M10K memory block[22] is a synchronous, true dual-port memory block, with registered inputs and optionally registered outputs, available in Arria V and Cyclone V family devices. The M10K block can be used for storing processor code, implementing lookup schemes, and implementing large memory applications. It can also be configured as true dual-port, simple dual-port, or single-port RAM and ROM.

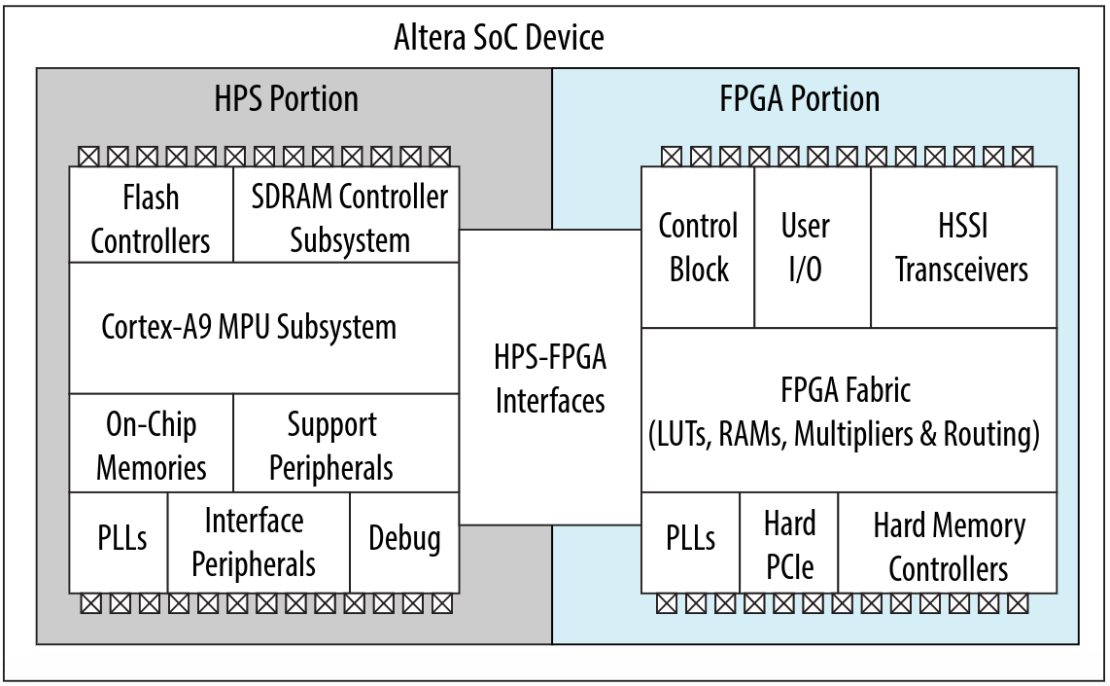


FIGURE 2.2: Cyclone V SOC-FPGA Overview
[21]

The Adaptive Logic Module (ALM)[23] is the basic building block of supported device families (Arria series, Cyclone V, Stratix IV, and Stratix V) and is designed to maximize performance and resource usage. Each ALM can support up to eight inputs and eight outputs, and contains two or four register logic cells, two combinational logic cells, two dedicated full adders, a carry chain, a register chain, and a 64-bit LUT mask.

2.1.7.2 Hard Processor System side

The Hard Process System (HPS) is a Cortex-A9 MPCore microprocessor integrated with a wide set of peripherals that reduce board size and increase performance within a system.

2.1.7.3 Cyclone V Interconnect

The Cyclone V SoC-FPGA provides a way for communication between the Programmable Logic side and the HPS side. The first and most basic bridge is the Lightweight HPS-to-FPGA bridge, which is designed for small transactions. Streaming data on this channel should be avoided. The limitation on size comes from the fact that the data bus of this bridge is only 32 bits wide. As the name of the bridge indicates, it is unidirectional

and goes from the HPS to the FPGA. Then come two more interesting bridges: the FPGA-to-HPS Bridge and the HPS-to-FPGA Bridge. The directions of these two buses are obvious. What is less obvious is their size. On the HPS side, their dimension is fixed at 64 bits, but on the FPGA side, they have the great advantage of being able to be configured to 32, 64, or 128 bits. These bridges are ideal for large data transfers. Finally, there are the FPGA-to-SDRAM bridges, which allow six masters on the FPGA side to have non-cached and non-coherent access to the RAM memory. These different bridges all allow several masters on one side to be connected to multiple slaves on the other. The selection of the slaves is done by byte-aligned addresses. The address space is therefore the only thing limiting the number of slaves on a bridge. In order to better visualize the interconnection, a diagram is provided in Figure 2.3.

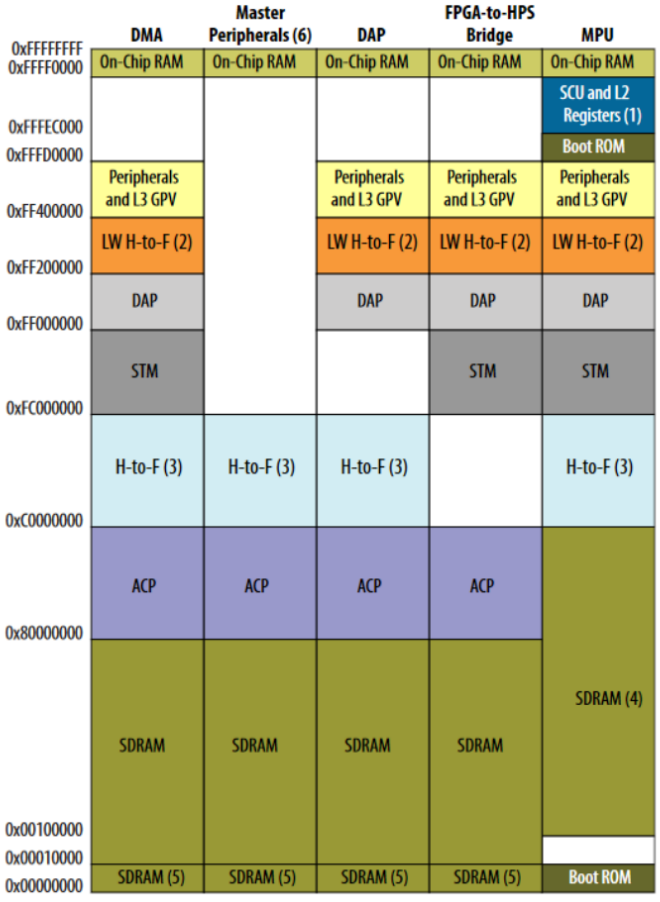


FIGURE 2.3: Address maps of Cyclon V interconnect [24]

2.2 Computer Network

A group of computers sharing resources that are available on or offered by network nodes is known as a computer network[25]. Over digital links, the computers communicate with one another using standard communication protocols. These connections are made up of telecommunication network technologies, which are based on physically wired, optical, and wireless radio-frequency means and may be set up in a number of different network topologies. Personal computers, servers, networking equipment, and other specialized or general-purpose hosts can all function as nodes in a computer network, while each node is identified by a network address.

2.2.1 Network Switches

A network switch is a piece of networking hardware that connects devices on a computer network by receiving and forwarding data based on the MAC address (Media Access Control). For high-performance applications, the number of ports and the speed of the switch are extremely important.

2.2.2 Ethernet Protocol

Ethernet is a family of wired computer networking technologies commonly used in Local Area Networks (LAN). It was commercially introduced in 1980 and first standardized in 1983 as IEEE 802.3[26]. Since then, Ethernet has been improved to accommodate faster data rates, more nodes, and longer link distances while retaining a significant amount of backward compatibility. Ethernet is commonly found in any high-performance application as its speed can reach up to 400 Gbps.

2.2.3 Internet Protocol

The Internet Protocol (IP)[27] is designed for use in interconnected systems of packet-switched computer communication networks. IP provides for transmitting blocks of data called datagrams from sources to destinations, where sources and destinations are hosts identified by fixed-length addresses called IP addresses. IP has two main functions:

- Addressing: The internet modules (i.e., hardware capable of networking) use the addresses carried in the internet header to transmit internet datagrams toward their destinations. The selection of a path for transmission is called routing.

- Fragmentation: Fragmentation of an internet datagram is necessary when it originates from a "large packet" network and must traverse a "small packet" network. The internet modules use fields in the internet header to fragment and reassemble internet datagrams.

2.2.4 Dynamic Host Configuration Protocol

The Dynamic Host Configuration Protocol (DHCP)[28] is a network management protocol that is used on Internet Protocol (IP) networks to automatically assign IP addresses to devices connected to the network via a client-server architecture. This technique removes the need to manually configure network devices. This is very convenient when setting up and deploying multiple network devices at the same time.

2.2.5 Transmission Control Protocol

The Transmission Control Protocol (TCP)[29] is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. TCP provides reliable inter-process communication between pairs of processes in host computers attached to distinct computer networks. The TCP protocol has two main functions: Reliability and Congestion Control.

2.2.5.1 Reliability

TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by a module. This is achieved by assigning a sequence number to each octet transmitted, and requiring a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. At the receiver end, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments.

2.2.5.2 Congestion Control

When a network node or link is transporting more data than it can process, it experiences network congestion, which results in a lower quality of service. Queueing delays, packet loss, and the blocking of new connections are all common side consequences. Network protocols such as TCP use aggressive retransmissions to compensate for packet losses.

However, this can lead to an increase in congestion, which causes more packet losses. This saturates the network card very quickly, which is a catastrophic problem known as "Congestive Collapse." To avoid this problem, TCP uses a network congestion-avoidance algorithm that combines multiple strategies, such as the Congestion Window, Slow Start, and Additive-Increase/Multiplicative-Decrease (AIMD) algorithm. All of these strategies combined are known as the TCP Congestion Control algorithm, described in RFC 5681 (Request for Comments)[30].

Chapter 3

Design and Implementation

The system's design objective is to give users access to a framework to develop applications for, as well as construct and control, a complex distributed system consisting of a Server and multiple Computing Nodes.

3.1 Top level system overview

The system consists of the following parts:

- Server
- SoC-FPGA Boards (Computing Nodes)
  - Node Manager
  - Hardware Cores
- Network Switch

These parts work together to create a distributed system consisting of various SoC-FPGA boards that parallelizes computations. Figure 3.1 shows a graphical representation of the system.

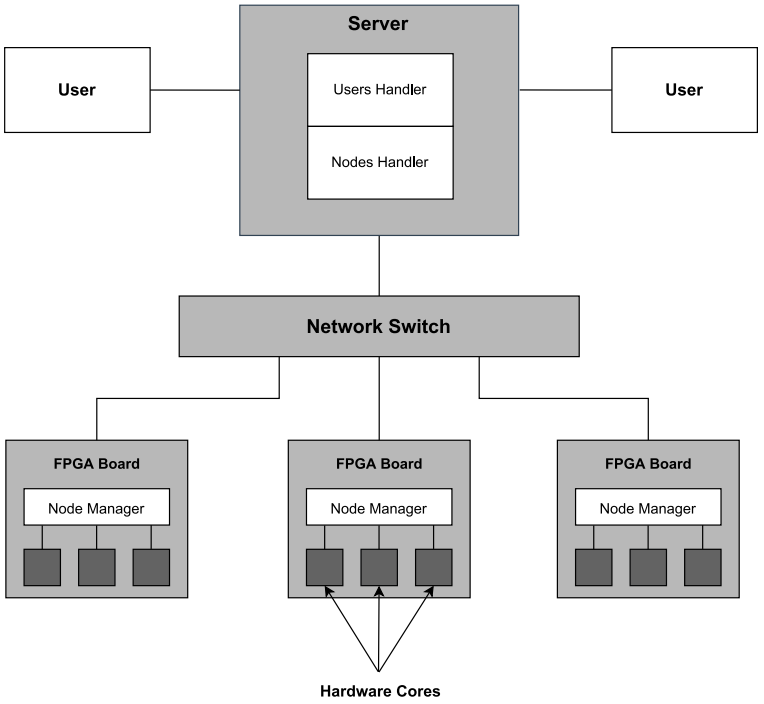


FIGURE 3.1: Graphical representation of the system.

In terms of operation, the Users Handler, which resides on the Server, provides the user with a set of functionalities to interact with the system. These functionalities can be used to connect to the Server, configure the Computing Nodes for different applications, send input data for computation, and receive resulting data. The user has the resources to create a software program that makes use of the FPGA boards by using these features.

The Nodes Handler, which resides on the other side of the Server, is another important part that controls the connected Computing Nodes and, more specifically, to the software running on each node. The Nodes Handler basically deals with the connection, the reconfiguration of the nodes, dispatching input data and collecting output data.

The Node Manager takes the form of a software application that runs on an Embedded Linux Operating System, which in turn runs on the HPS side of the SoC-FPGA. This application is the heart of the Computing Node, and it has full control over the programmable logic side (configuration, read, and write). The Node Manager serves as a middleware between the Server and the Hardware Cores, which are logic blocks that perform specific computations. This is due to the fact that the Node Manager is responsible for routing data received from the Server to the Hardware Cores, retrieving the processed data, and send it back to the Server.

The Server and FPGA boards are connected together in a local network via a gigabit Ethernet Switch to allow for communication between the Server's software application and the Node Manager software of the Computing Nodes. The user program can also

be connected to the same network as the Computing Nodes or to a different network. A custom messaging protocol was designed to be added on top of the TCP/IP protocol to simplify data flow between different components of the system.

3.2 Serial Messaging Protocol

Networking is an essential part of the project. The different hardware components should be able to communicate with each other in a reliable and error-free way. For this reason, the base transport protocol chosen is the TCP over IP protocol which provides reliability and flow control. However, a custom messaging protocol, that we named: Serial Messaging Protocol (SMP), was implemented to extend these functionalities and address the needs of the project.

The Serial Messaging Protocol (SMP) is a simple protocol that was designed to simplify the transfer of configuration and data messages across the different components of the system. Since this custom protocol does not provide any means of error correction and flow control, the protocol should be based on the TCP/IP protocol or any transport protocol that provides these functionalities. The "Serial" and "Messaging" in SMP imply that communication is achieved using concrete messages that are serialized into the TCP's transmit and receive streams. This means that only one message can be sent in a given stream at a time.

3.2.1 SMP Generic Message

An SMP message consists of a header of eight bytes and a variable length payload. Figure 3.2 shows the format of a generic SMP message. The first Byte of an SMP message determine what type of message it is. There are three types of messages:

- Configuration Message (Type 1).
- Configuration Response Message (Type 2).
- Data Message (Type 3).

The three bytes after the type field are reserved for future use and are mostly there to keep the header aligned to four bytes to improve message processing on 32-bit machines. The Payload Size field is four bytes long and represents the size of the payload in the message without counting the eight bytes of the header. This field should be interpreted as a 32-bit unsigned integer in little-endian byte order. This implies that the maximum payload size is 4 GB.

All SMP messages contain the same header format. However, they differ in the payload format.

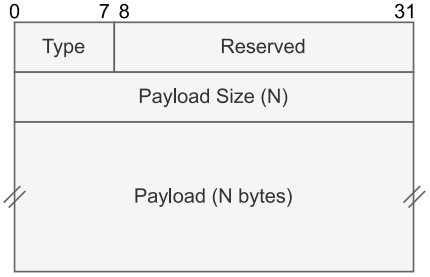


FIGURE 3.2: SMP Generic Message format.

3.2.2 SMP Configuration Message

The payload of the Configuration Message contains the Bitstream Raw Binary File (RBF) that is used to configure the Programmable Logic part of the Computing Nodes. Along with this, it also contains how many Hardware Cores there are in the Bitstream RBF, information about the base addresses of each Hardware Core, and the expected size of input and output data that the cores are designed for.

Figure 3.3 shows the format of a Configuration Message. The first twelve bytes of the payload hold the Input Data Size, Output Data Size, and Number of Hardware Cores. These three values are four bytes each and should be interpreted as unsigned 32-bit integers in little endian byte order. Followed by M Hardware Core Information blocks, with each consisting of a Control Base Address, an Input Base Address, and an Output Base Address, while each address field is four bytes long. The remaining bytes of the payload contain the Bitstream RBF.

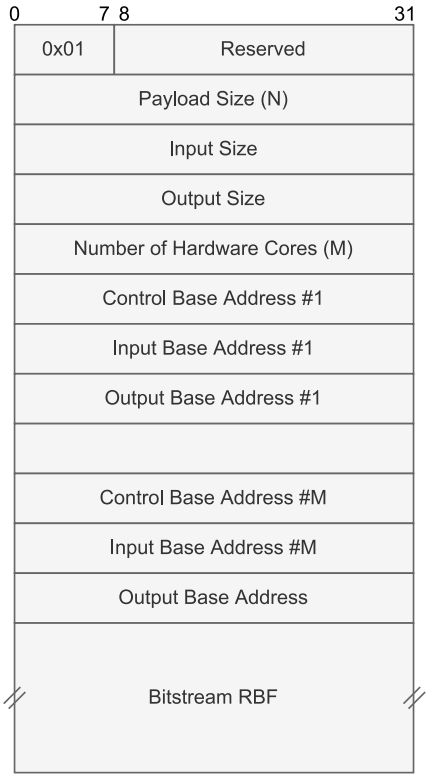


FIGURE 3.3: SMP Configuration Message format.

3.2.3 SMP Configuration Response Message

The Configuration Response Message is used to respond to a received Configuration Message. The first byte of the payload holds the Status of the configuration. A zero valued status indicates a successful configuration, while a non-zero valued status indicates a failure. Figure 3.4 shows the format of the Configuration Response Message.

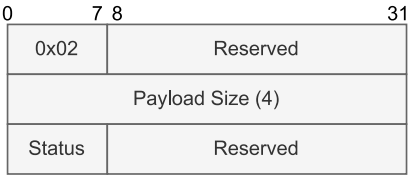


FIGURE 3.4: SMP Configuration Response Message format.

3.2.4 SMP Data Message

The type 3 SMP message, the Data Message, is used to move data between two connected peers. The first four bytes of the payload hold the ID of the data being sent. This field can be used to pair a data message with its corresponding result data message. followed by an arbitrary-sized data field whose size can be determined by subtracting four bytes from the payload size field of the header. Figure 3.5 represents the format of the SMP Data Message.

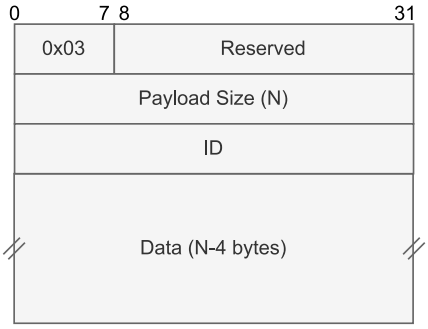


FIGURE 3.5: SMP Data Message format.

3.3 Server

The server is the central part of the implementation, it sits between the user and the Computing Nodes. There are four key tasks that the server is responsible for:

- Managing and assigning Computing Nodes to users.
- Configuring Computing Nodes with the Hardware Configuration received from the user.
- Routing the data between the user and the Computing Nodes assigned to it.
- Balancing the workload distributed among Computing Nodes.

The server is implemented in C++ using the ASynchronous Input/Output (ASIO) library, which is a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach. The library also provides support for C++20 coroutines, which allows developers to write asynchronous code in a synchronous way.

The server listens for TCP connections on two different port numbers, one for node connections and another one for user connections. The server, user software, and nodes all use the SMP protocol described in the section 3.2 for communication. When a Computing Node connects through the nodes port, the connection is held open and kept in the unassigned nodes queue. Once a user connects, a User Handler is created and the user connection gets assigned to it along with all the node connections available in the unassigned nodes queue. Figure 3.6 shows the top-level architecture of the server.

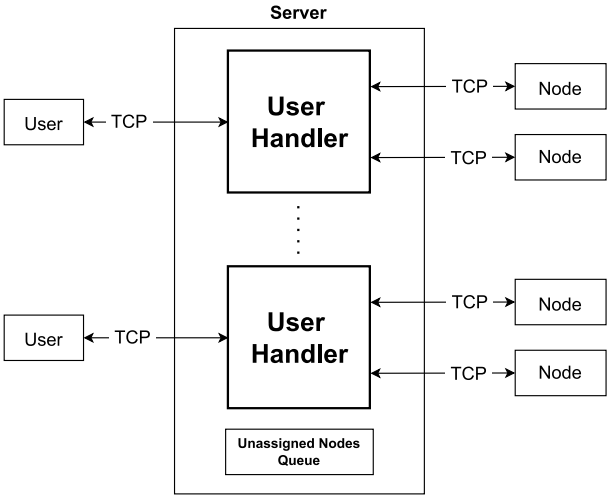


FIGURE 3.6: Server top-level architecture diagram

The server supports multiple users with multiple nodes assigned to them. However, this implementation focuses on the User Handler logic rather than managing multiple users. Therefore, for the rest of this section, we assume that only one user is connected to the server and all of the connected Computing Nodes are assigned to that user.

3.3.1 User Handler

The User Handler is a coroutine responsible for configuring and managing the Computing Nodes assigned to it and routing the data between the user and the Computing Nodes.

As shown in Figure 3.7, once the User Handler is created, it starts in the Initial State. When the first Configuration Message is received from the user, it changes to the Nodes Configuration State where the User Handler configures all of the Computing Nodes with the received configuration. And when the Computing Nodes configuration is finished, the User Handler enters the Data Routing State where Data Messages received from the user are distributed to the Computing Nodes and the result Data Messages received from the Computing Nodes are sent to the user. If a Configuration Message is received while the User Handler is in the Data Routing State, the state is changed back to the Nodes Configuration State.

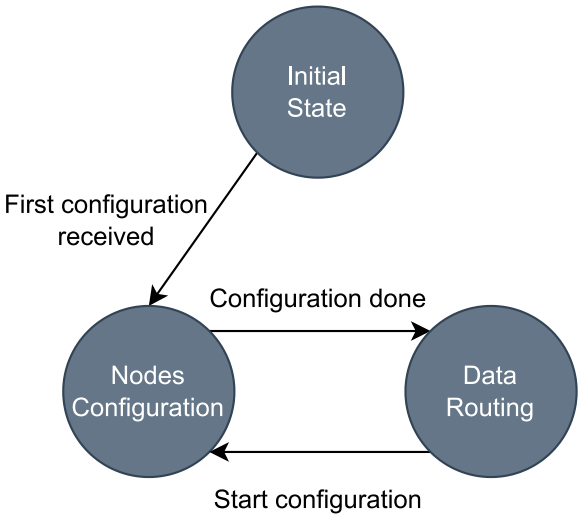


FIGURE 3.7: User Handler state machine diagram.

3.3.1.1 Initial State

When the User Handler is created, it waits for an SMP message from the user. The first message should be a valid SMP Configuration Message, otherwise, an error is reported back to the user and the User Handler together with the associated Computing Node connections are dropped. If the received message is a valid SMP Configuration Message, the configuration is saved and the state is set to the Nodes Configuration State as seen in Figure 3.8.

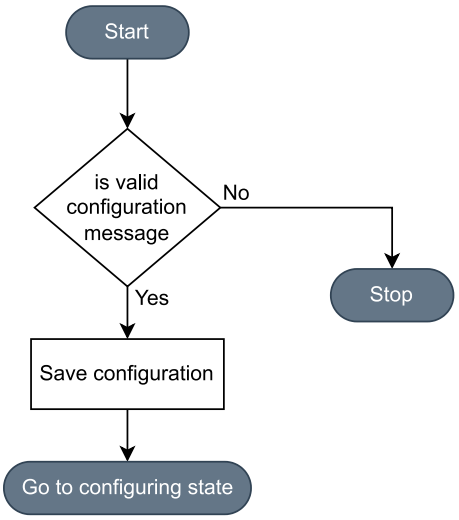


FIGURE 3.8: User Handler Initial State flowchart.

3.3.1.2 Nodes Configuration State

When a valid SMP Configuration Message is received, the User Handler enters this state and proceeds to configure all the Computing Nodes assigned to it. At this state, it is assumed that a valid configuration is stored locally and is the one used to configure the nodes. The flowchart shown in figure 3.9 describes the logic of the Nodes Configuration State.

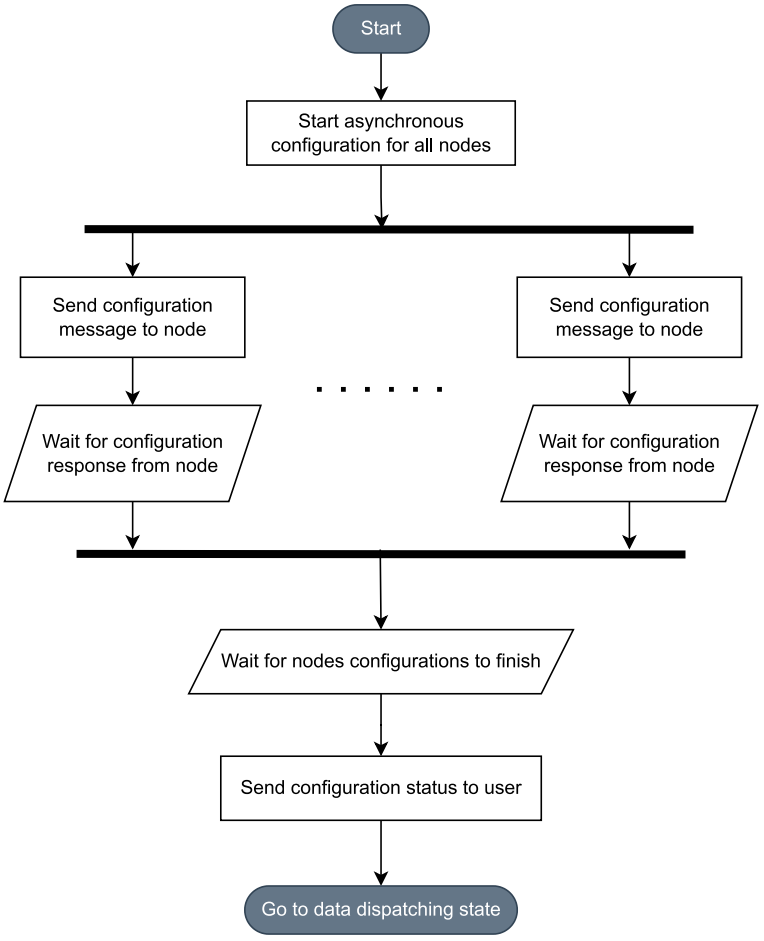


FIGURE 3.9: User Handler Nodes Configuration State flowchart.

As shown in the flowchart, the User Handler starts an asynchronous operation for every Computing Node assigned to it. The asynchronous operation consists of forwarding the SMP Configuration Message to the Computing Node and waiting for an SMP Configuration Response Message back from the Computing Node. If an error occurs on the connection or a negative SMP Configuration Response Message is received, the

node connection is assumed to be broken and the Computing Node is removed from the nodes list of the User Handler. After these asynchronous operations are started, the User Handler waits for all of them to finish. A positive SMP Configuration Response Message is sent to the user if at least one Computing Node is configured successfully. Otherwise, a negative SMP Configuration Response Message is sent.

3.3.1.3 Data Routing State

The Data Routing part of the User Handler is considered the most important part since all the data traffic moves through it. This implies that the overall throughput and latency are affected by the architecture of this part. The data flow from the user connection to the computing nodes and the result flow from the computing nodes to the user connection are depicted in Figure 3.10. The SMP Data Messages received from the user are routed to the Data Demultiplexer, which determines which computing node they should be routed to. On the other hand, for every Computing Node assigned to the User Handler there is an asynchronous receiver loop assigned to it, which listens for incoming SMP Data Messages received from the Computing Node, checks their validity, and forwards them to the user.

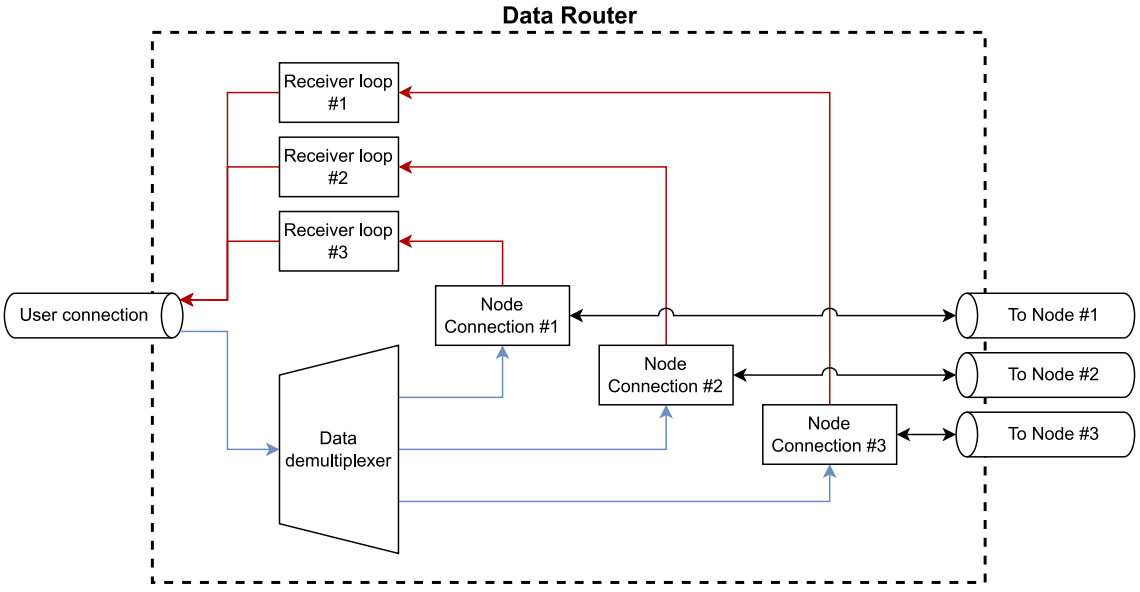


FIGURE 3.10: Data Routing data flow diagram.

The Data Demultiplexer is responsible for selecting which Computing Node receives the Data Message. This is considered as a hot path (i.e. this path has a large amount of traffic). Therefore, the time complexity of the algorithm should be as low as possible to achieve high throughput and low latency. This is achieved by making sure no copy

operation occurs after a Data Message is received from the user, which is done by only moving around pointers to the data. The space complexity is not considered an issue as it is proportional to the number of associated Computing Node connections only, which by itself has a minimal memory footprint.

Two implementations have been realized for the Data Demultiplexer: Stack-based and Queue-based. Both implementations have four common steps:

- Receive one SMP Data Message from the user.
- Pop one Computing Node connection from the stack/queue.
- Forward the SMP Data Message to the popped Computing Node connection.
- Push back the Computing Node connection to the stack/queue when the sending is done.

Figures 3.11 and 3.12 show diagrams of the Stack-based and Queue-based implementations, respectively.

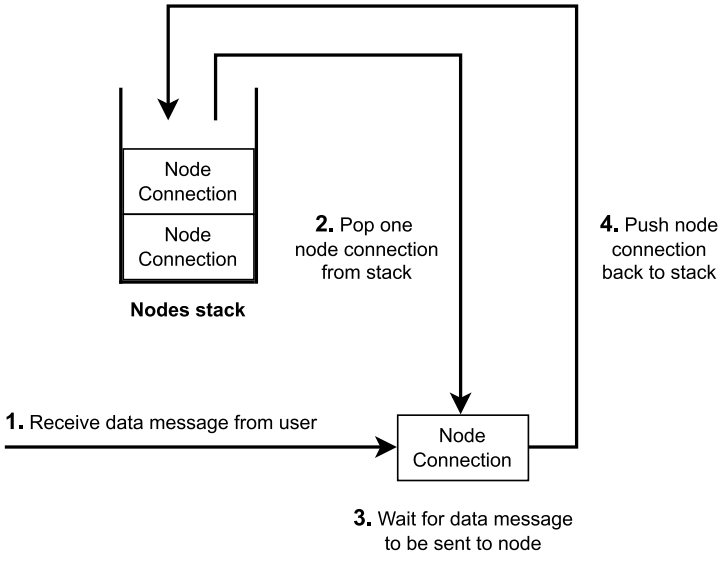


FIGURE 3.11: Stack-based Data Demultiplexer diagram.

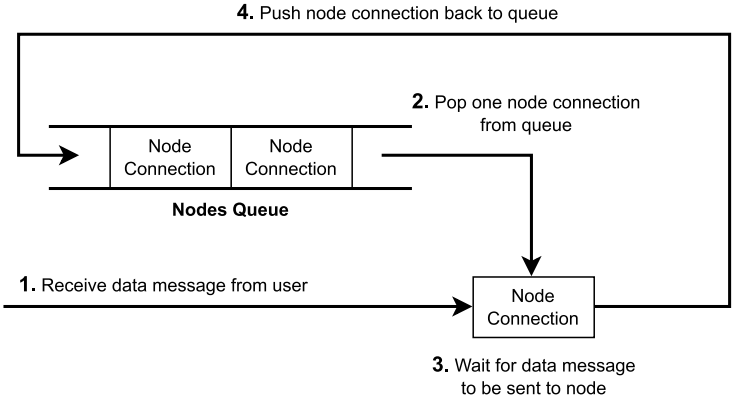


FIGURE 3.12: Queue-based Data Demultiplexer diagram.

Both implementations rely heavily on the TCP Congestion Control algorithm described in section 2.2.5.2. The TCP sending request will block if the remote Computing Node does not consume the messages sent previously and its TCP receive buffer limit is reached. This behavior allows the Data Demultiplexer to slow down the sending rate of the messages as any popped Computing Node connection is not pushed back to the nodes stack/queue until the sending is successful. These implementations result in two different behaviors: For the Stack-based implementation, the load will be heavily concentrated on the top nodes of the stack. The nodes closer to the bottom of the stack will get Data Messages sent to them only if the top nodes block when sending data. On the other hand, the Queue-based implementation distributes the data to the nodes in a Round-Robin fashion[31], but not exactly in the same order each time. i.e., after sending the SMP Data Message, the popped Computing Node connection is pushed to the back of the queue, and the next connection in the front of the queue is popped for the next Data Message to be sent. This process repeats for every Data Message received from the user.

The Data Router also holds a Data Counter, which is used to keep track of how many Data Messages were sent to the Computing Nodes without having their corresponding result Data Messages received. The Data Counter is incremented for every Data Message sent to any of the nodes and decremented for every result Data Message received. Another important part of the Data Router is the Result Receiving Loop, which is also considered to be a hot path. It is a simple and straightforward coroutine, and each Computing Node connection gets assigned a Receiver Loop to handle the result Data Messages received and forward them to the user.

Figure 3.13 shows a flowchart of the Result Receiving Loop. The loop waits for a Data Message from the Computing Node, checks its validity, forwards it to the user, and decrements the Data Counter.

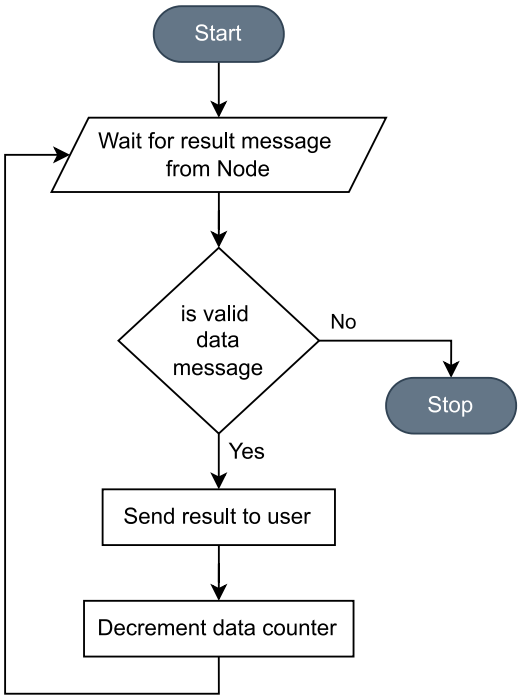


FIGURE 3.13: Result Receiving Loop flowchart.

Figure 3.14 shows a flowchart of the Data Routing State logic. When the User Handler enters this state, it first starts the Result Receiving Loops for all the Computing Nodes. Then it waits for an SMP Data Message from the user, and if the received message is valid, one node connection is popped from the stack/queue, and an asynchronous sending operation is started on the node connection. The asynchronous sending operation consists of waiting for the received Data Message to be sent to the Computing Node, incrementing the Data Counter, and pushing the Computing Node connection back to the stack/queue. At the same time, the main Data Routing coroutine goes back to listening for incoming messages from the user.

In the case where the received message from the user is an SMP Configuration Message, the User Handler stores the configuration and waits for the Data Counter to drop to zero, indicating that there are no pending operations on the nodes. After that, the nodes are assumed to be idle and the Result Receiving Loops are cancelled. Waiting for the Data Counter to drop to zero is essential when reusing the same nodes with a new Hardware Configuration.

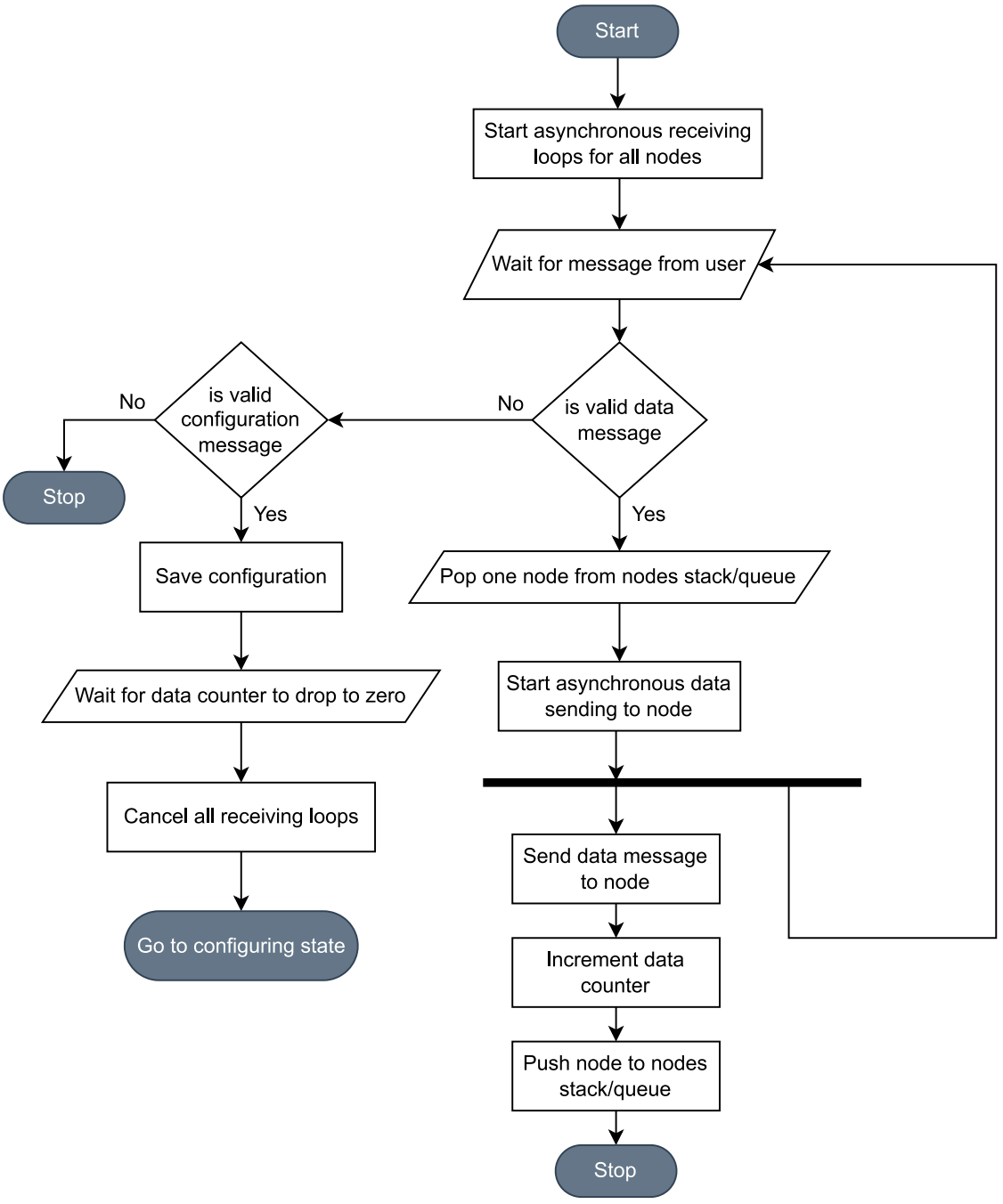


FIGURE 3.14: Data Routing flowchart.

3.4 Computing Node

The Computing Node is a development board (DE10 SoC-FPGA [18]) that contains a Cyclone V SoC-FPGA chip, which in turn contains two distinct parts: a Hard Processor System (HPS) and a Programmable Logic (PL). The HPS runs an Embedded Linux Operating System which hosts a managing software referred to as the Node Manager. The PL on the other hand, will be programmed with different hardware designs. The programmed hardware design contains Hardware Cores that process data for a specific application. Before the Server sends any data, an SMP Configuration Message should be sent first to configure the PL. The Node Manager is responsible for communicating with the Server, configuring the PL, distributing the data received from the Server to the Hardware Cores inside the PL and retrieving the results and sending them back to the Server. Figure 3.15 shows the Computing Node internal architecture.

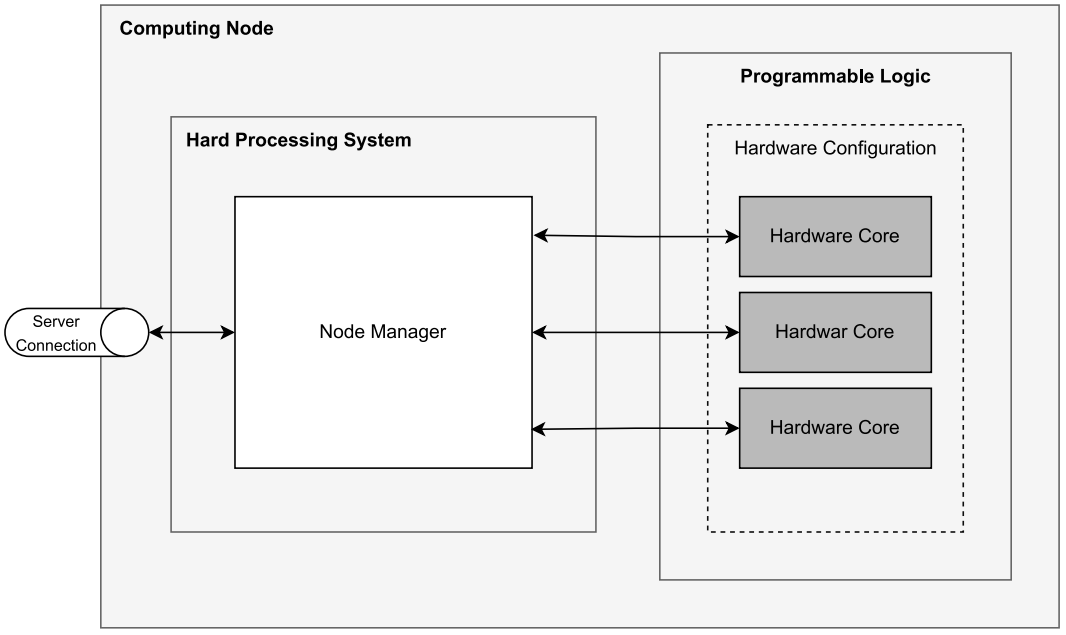


FIGURE 3.15: Computing Node Architecture

3.4.1 Hardware Configuration

A Hardware Configuration is the combination of a Bitstream Raw Binary File that the Node Manager uses to configure the PL, and information about the Hardware Cores inside the hardware design.

The following subsections explain how the Node Manager interacts with the hardware design programmed on the PL and how a new Hardware Configuration is generated.

3.4.1.1 Node Manager-Hardware Cores communication

The Node Manager software on the HPS is responsible for sending the data to the Hardware Core, triggering the start of processing and retrieving the resulting data. This implies that the Hardware Core should provide a memory interface for input data, output data and control registers.

The Intel Cyclone V chip contains multiple AXI bridges to allow communication between the HPS and the PL. Since the Node Manager controls the Hardware Cores, only the HPS-to-FPGA bridge is used, where the HPS is the bus master. As shown in Figure 2.3, the HPS-to-FPGA bridge is mapped to the memory region 0xC0000000-0xFC000000. The Hardware Cores are then distributed across this memory region.

From the software point of view, the Linux system call function `mmap()` creates a mapping of the HPS-to-FPGA bridge's physical address space to the virtual address space of the application process.

3.4.1.2 Hardware Cores

The Hardware Cores are the last components in the system tree. They are the components that perform the data processing on the data sent by the user and generate result data to be sent back. As mentioned in the previous section, a Hardware Core should provide interfaces for three memory blocks local to it:

- Input: this memory block is used to store the data that should be processed.
- Output: used to store the resulted data.
- Control: provides an interface to start the processing and retrieve information about the status of the processing.

As an example, a Hardware Core that performs the addition of two (N) elements arrays (1 Byte signed integer elements), the Input and Output memory blocks' sizes should be 2x(N) and (N), respectively.

Since the Node Manager is a non-configurable part and always running on the HPS, the Control memory block layout should be standardised for different Hardware Cores implementations. i.e. a new design of a Hardware Core should follow the same layout for its Control memory block.

HLS is used for Hardware Cores development because it automatically generates a Control memory block called the Control and Status Registers (CSR) which is used

to control the Hardware Core and retrieve its status information. Furthermore, the Hardware Core design is synthesised into an RTL design that can be simulated for functional and timing analysis. Table 3.1 shows the different registers of the CSR.

Address	Access	Register Contents	Description
0x00	R	reserved (62:0) busy 0:0	Read the busy status of the component 0 - Ready to accept a new start 1 - Cannot accept a new start
0x08	W	reserved (62:0) start (0:0)	Write 1 to signal start to the component
0x10	R/W	reserved (62:0) interrupt_enable (0:0)	0 - Disable interrupt 1 - Enable interrupt
0x18	R/Wclr	reserved(61:0) done(0:0) interrupt_status(0:0)	Signals component completion done is read-only and interrupt_status is write 1 to clear

TABLE 3.1: Control and Status Registers memory layout.

The following code shows the HLS component function signature used to generate a Hardware Core that is compatible with the system and that the Node Manager is designed to work with:

```
hls_avalon_agent_component
component void hardware_core(
            hls_avalon_agent_memory_argument(INPUT_SIZE) uint8_t* in,
            hls_avalon_agent_memory_argument(OUTPUT_SIZE) uint8_t* out);
```

The `hls_avalon_agent_component` attribute at the start of the component signature tells the compiler to generate the CSR. The component should take only two arguments preceded with the `hls_avalon_agent_memory_argument(N)` to tell the compiler to interpret the argument as a memory array of N elements. One argument is for Input data and the other is for Output data. When compiling a component with this signature, the HLS compiler generates an IP package for it that contains three avalon interfaces and that can be easily integrated into the Intel Platform Designer to connect it to the HPS through the HPS-to-FPGA AXI bridge.

Figure 3.16 shows the interconnect between the Node Manager running on the HPS and the Hardware Cores on the PL side. Since the HLS compiler generates the memory interfaces as Avalon interfaces, the Intel Platform Designer adds an Avalon to AXI interconnect logic to bridge the two buses.

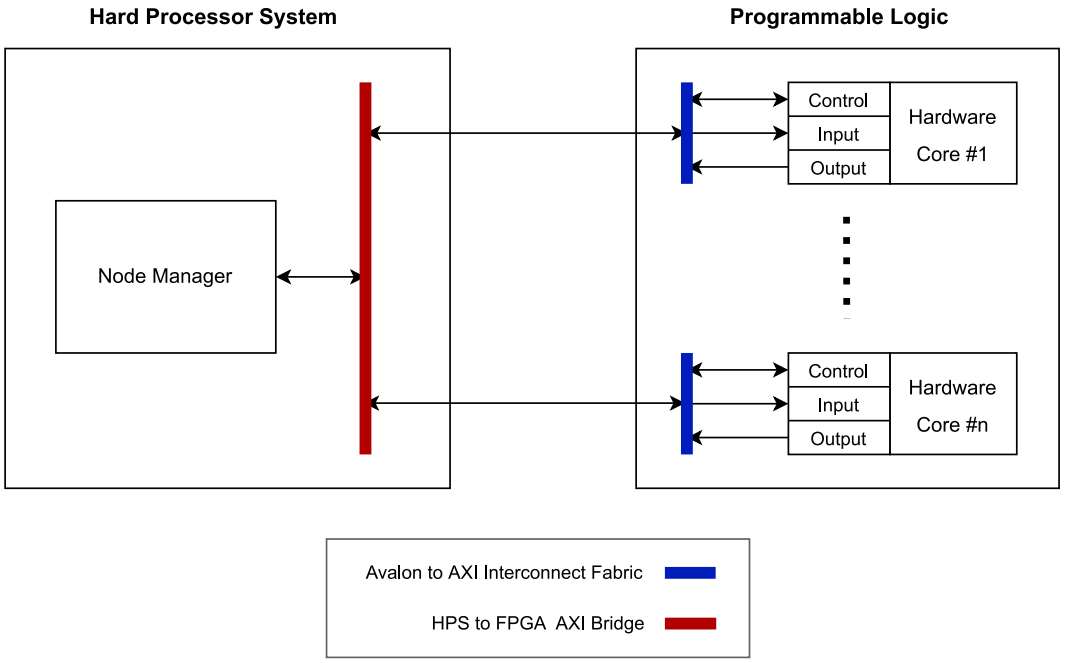


FIGURE 3.16: Node Manager to Hardware Cores interconnect.

3.4.1.3 Configuration generation

A template Quartus project has been set up to simplify the integration of a custom Hardware Core provided by the user into a new Bitstream RBF that is compatible with the system. The newly generated Bitstream RBF together with metadata information about the Hardware Cores are referred to as a Hardware Configuration. An SMP Configuration Message is then created using this Hardware Configuration, which can be sent by the user to configure the Computing Nodes.

3.4.2 Node Manager

The Node Manager is a software application. Its architecture is based on two concepts, Multi-threading and Data Queues. It consists of the following components

- Receiving Thread (Main)
- Sending Thread
- Input Data Queue
- Output Data Queue
- Core Handler Thread Pool

The first thread is the Main Thread, its job is receiving messages from the server, initialization and configuration of the Programmable Logic, creating and controlling the worker threads, dealing with receiving and organising data. The second thread is for sending results back to the server. The two Data Queues, one for Input Data and the other for Output Data, act like a bridge between the Main and Sending Thread with the Core Handlers. These queues provide thread safe read and write operations. Another important part of the architecture is the Core Handler Thread Pool. The number of Core Handlers depends on the number of Hardware Cores as each Hardware Core is assigned to a unique Core Handler. Each Core Handler is mainly responsible for routing Input Data from the Input Data Queue to the Hardware Core and vice versa for the Output Data. The architecture design of the Node Manager is shown in Figure 3.17.

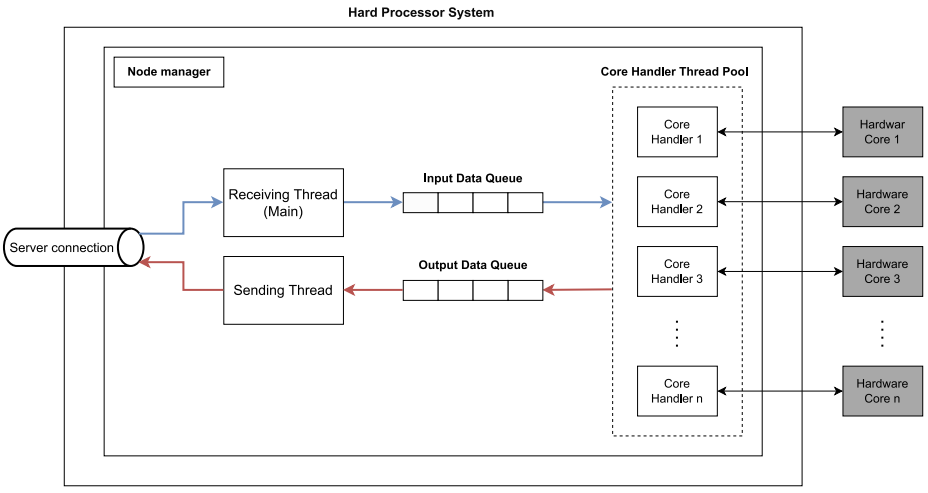


FIGURE 3.17: Node Manager architecture

3.4.2.1 Operating system

The Node Manager operates on top of an open source minimal Embedded Linux OS called Rsyocto[32], the operating system is dedicated for Altera SoC-FPGA boards. Rsyocto supplies multiple features for both development and operation phases.

3.4.2.2 Operation

The Node Manager application conducts these essential operations:

- Initialization and connection
- Run-time reconfiguration
- Data routing
- Memory space tracking

Figure 3.18 is a flowchart that describes the operation of the Node Manager which includes the configuration and the computation process.

At startup, the Embedded Linux launches the software application through a startup script. On the Main Thread of the application multiple variables and data structures are initialised alongside with the memory mapping of the HPS-FPGA and the lightweight HPS-FPGA bridges. A TCP/IP connection is then established with the Server. Afterwards, The Main Thread waits for the first SMP Configuration Message from the server.

The flexibility of the system is highly dependable on its ability to be reconfigured with any Hardware Configuration at run-time. The reconfiguration process is triggered when the Node Manager receives an SMP Configuration Message described in section 3.2.2 from the server.

This process starts by clearing every Core Handler from the thread pool and setting the Node Manager State to Stop State.

Then, the Node Manager proceeds to configure the Programmable Logic with the Bitstream RBF encapsulated inside the Configuration Message. The Node Manager will also report back whether this operation succeeded or not.

Next, The Node Manager will use the remaining information in the SMP Configuration Message to instantiate the Core Handlers. This information represents a detailed description of the design and the number of Hardware Cores and their base addresses. Once the Core Handlers are instantiated, each Core Handler will store the base addresses of their corresponding Hardware Core to enable read and write operation. These operations are as simple as basic memory operations as they can be done either by using simple pointers or by using a C standard function called `memcpy()`.

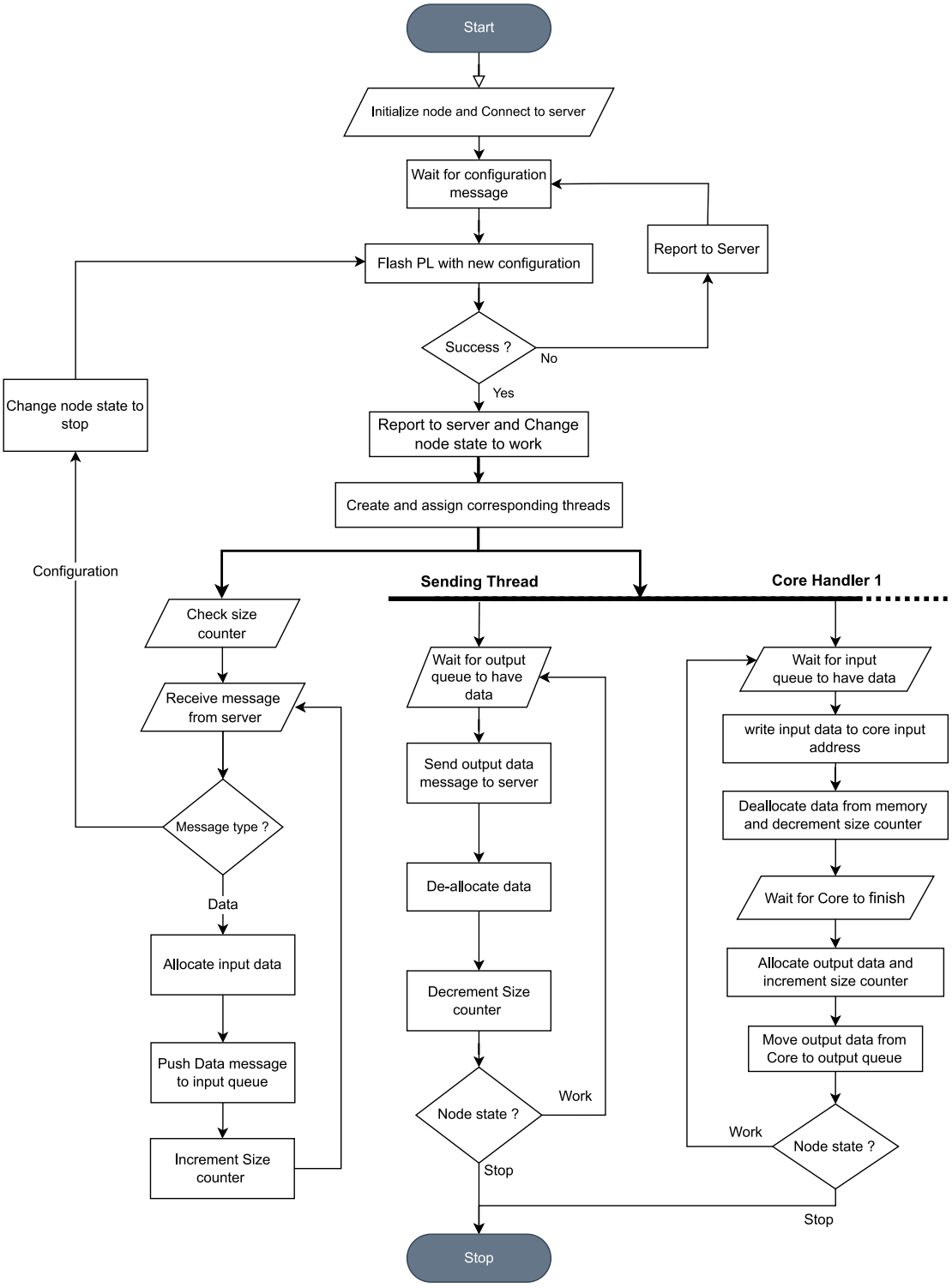


FIGURE 3.18: Node Manager Application Flowchart

In the final stage of configuration, the Main Thread becomes the Receiving Thread and a Sending Thread is also started. In the end, The Node Manager State is set to Work State.

Once the initial configuration is finished, the Node Manager waits for SMP messages from the server. In the case where the received message is an SMP Configuration Message, The Node Manager State is set to Stop State and the same configuration process that occured in the initial configuration will start.

Assuming the received message is an SMP Data Message, which is described in Section 3.2.4, the Node Manager proceeds to heap allocate memory of size equal to the size of the Data field of the SMP Data Message and copy the content of this field to the newly allocated memory. In this case, The Data field is considered as Input Data. Afterwards, the Node Manager constructs a data structure called `DataChunk` that contains a pointer to the newly allocated memory which represents Input Data paired with the same ID contained in the received SMP Data Message. Since copying is an expensive operation when dealing with large data, pointers are used instead of the Input Data itself. This data structure is then pushed into the Input Data Queue.

Next, any arbitrary Core Handler can pop a `DataChunk` structure from the Input Data Queue. Afterwards, the Core Handler proceeds to send the Input Data to its corresponding Hardware Core. Once finished, the Input Data is deallocated and deleted from memory. Figure 3.19 shows the lifespan of the Input Data inside the Node Manager.

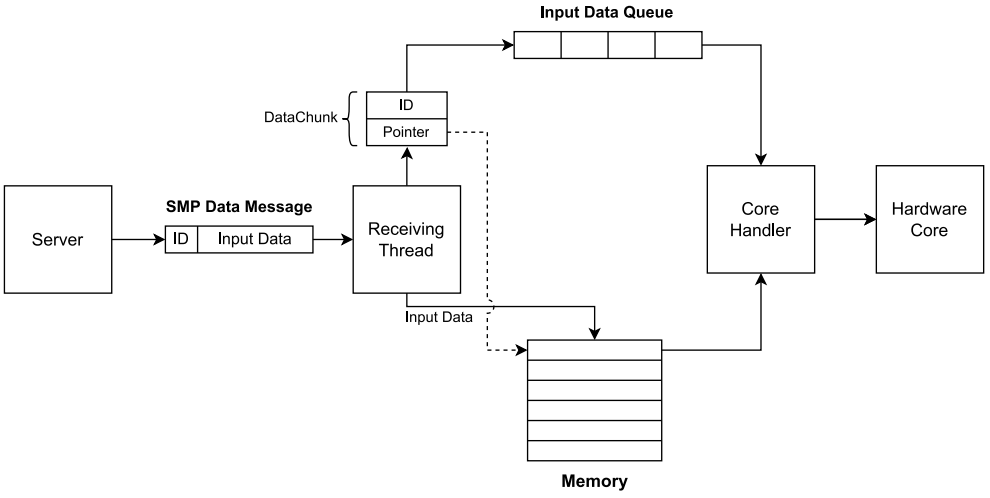


FIGURE 3.19: Input Data Flow Diagram

Once the Hardware Core finishes computation, The Core Handler proceeds to heap allocate memory equal to the size of the Output Data sent by the Hardware Core and

copy its content to the newly allocated memory. Afterwards, the Core Handler constructs a `DataChunk` containing a pointer to the Output Data in memory paired with the same previous ID. This `DataChuck` is pushed into the Output Data Queue. Then, the Sending Thread proceeds to pop a `DataChunk` from the Output Data Queue and constructs an SMP Data Message containing the Output Data and ID. Once this message is sent to the server, Output Data is deallocated and deleted from memory. Figure 3.20 shows the lifespan of the Output Data inside the Node Manager.

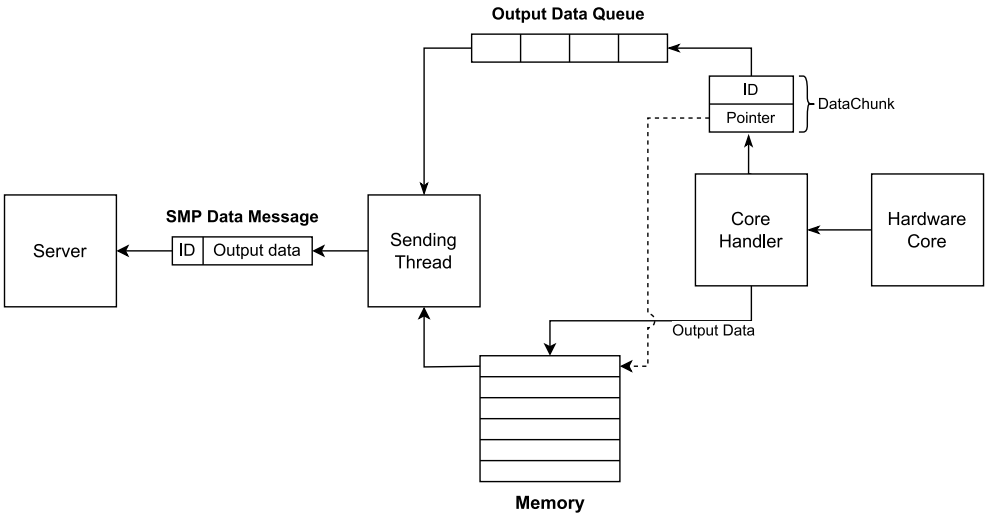


FIGURE 3.20: Output Data Flow Diagram

Memory management is extremely important inside the Node Manager as the Random Access Memory (RAM) of DE10 SoC-FPGA [18] is only 1 GB. In other words, Tracking memory allocations is necessary. This is done through the Size Counter which is a global variable that can be accessed by every component of the application. It works like a guard for the Node Manager memory. The Size Counter makes sure that memory usage never gets close to the maximum size of 1 GB. Receiving data from the server is always dependable on the space left in memory. Each time a memory allocation and deallocation is performed, the Size Counter is incremented and decremented respectively.

For maximum performance, whenever a thread is waiting for resources, it yields back the processor to other threads to work.

Chapter 4

Results and Discussion

4.1 Overview

Besides designing and implementing the distributed system alongside its framework, our aim was to assess each of its different components in a performance based analysis to determine the overhead of the system on the user application. One way of testing the performance of a system is calculating its throughput and latency. Throughput is a measure of how many units of information a system can process in a given amount of time[33] and it is defined by Equation 4.1.

$$Throughput = \frac{Bits}{Second} \tag{4.1}$$

Bits represents input data (total amount of data to be processed) and Second represents the application runtime. Another variable we relied on when testing phase was the latency of the system which has a great impact on real-time applications. Latency is the time required in a network for the one-way or round-trip transfer of data between two nodes[34].

Finally, we conducted a comparison of the efficiency of the system against a software-based implementation of a test-case application.

The system testing and the software based implementation were conducted on a machine with an Intel Core i7 processor and 8GB of RAM running Linux kernel version 5.18.7.

4.2 Load-free System Benchmark

4.2.1 Messaging Protocol Benchmark

A simple echo server was implemented that uses the SMP protocol for communication to analyse the performance of the networking stack. The performance analysis of the protocol is conducted mainly to get a reference performance model to compare it against the Server performance.

The echo server is a simple server that listens for incoming client connections and runs asynchronously a client handler, which in turn listens for incoming messages and echos them back to the client. A client program is also implemented, which connects to the echo server and starts sending messages with random data and with a fixed payload size. The client runs for ten seconds and calculates the average throughput and logs the information to the console for the analysis.

A Python script was written to automate the process of launching different numbers of clients with different payload sizes and parsing the console output of the clients to create a table that is used to plot a performance chart. The Python script runs the benchmarks with 1, 2, 3, and 4 concurrent clients. The benchmark starts with a payload size of 1 byte and doubles the size in every iteration up to a 2 Gigabyte payload size.

Figure 4.1 shows the total throughput versus the payload size with different number of clients.

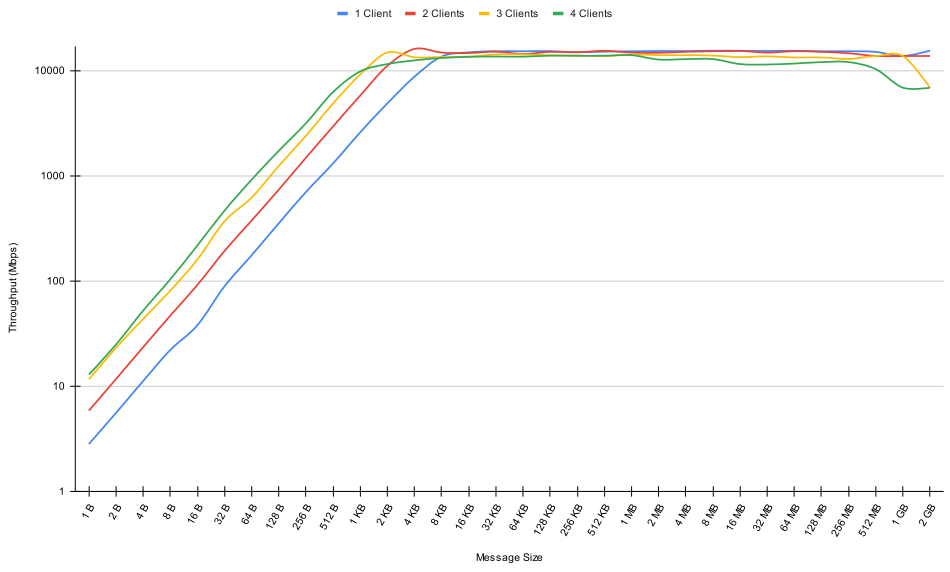


FIGURE 4.1: SMP echo server throughput benchmark

From the graph, we can detect two phases of the test. The first phase is from 1 byte to 4KB payload size, where the throughput follows an increasing pattern, reaching 16 Gbps as a maximum. In the second phase, from 4 KB to 2 GB payload size, the throughput is stable at a maximum of 16 Gbps.

These obtained results represent the maximum throughput a server implementation can get on the machine used for testing.

4.2.2 Server Benchmark

To allow the user to do a benchmark on the system, a custom SMP Configuration Message referred to as the Benchmarking Configuration was created, which can be detected by the Computing Nodes and switched to benchmarking mode. In this mode, the Computing Nodes do not program the PL side of the SoC-FPGA but rather simulate a computation delay provided in the Benchmarking Configuration. i.e. The Computing Node does not perform any processing on the input data and generates random output data with the specified time delay in between.

Figure 4.2 shows the format of the custom Benchmarking SMP Configuration Message. The Computing Nodes detect the Benchmarking Configuration when the Number of Hardware Cores is zero and the Payload Size is 16 bytes.

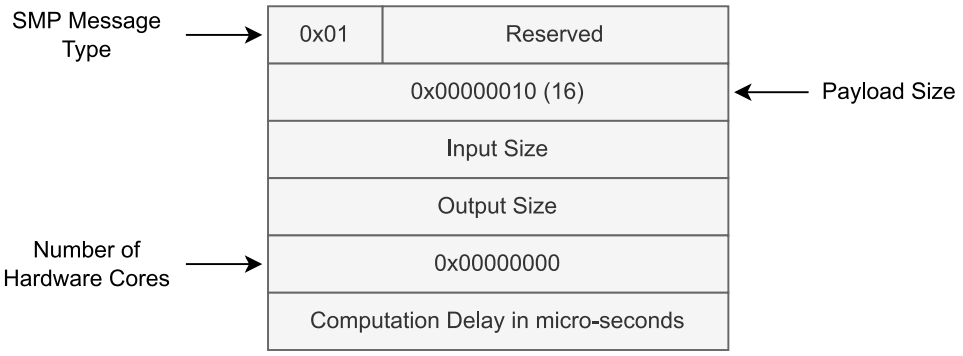


FIGURE 4.2: Benchmarking SMP Configuration Message Format

The Benchmarking Configuration can contain arbitrary Input and Output sizes, and the size of the Bitstream RBF field is four bytes long and is used to store the simulation processing time delay. Additionally, when the Computing Node is in the benchmarking mode, it copies the first sixteen bytes of the data field of a received Input Message into the first sixteen bytes of the data field of its corresponding Output Message. This allows the user to store 64 bits of data in an Input Message and retrieve it back from its corresponding Output Message. This feature is primarily added to store the

creation timestamp of an Input Message and calculate the round-trip latency when its corresponding Output Message is received. This implies that the minimum Input and Output sizes are sixteen bytes.

Furthermore, a simulation node is implemented which can be run on the same machine as the Server. This allows testing the Server performance with no networking hardware limitations since the user program, Server and simulation nodes are all connected to the loopback network interface of the operating system. The simulation node is designed to work exactly like the Computing Node but only in benchmarking mode.

A benchmarking user program is also implemented, which connects to the Server and sends a Benchmarking Configuration. The program runs for ten seconds in which it sends random input data, calculates the average throughput and latency, and logs the data to the console.

A Python script was written to automate the benchmarking process and observe the server's behaviour with different benchmarking configurations. The script launches in parallel the Server program, the benchmarking user program and the simulation nodes for all the combinations of input sizes and input counts, starting with a sixteen bytes input size and one input count, up to a 128 KB input size and 16384 input count in powers of 2. Hence, the maximum benchmarked Input Message is 2 GB. Both the Queue-based and the Stack-based implementations of the Server are tested and compared.

To preserve the integrity of the benchmark, the maximum number of concurrent simulation nodes that can be run is four, because the machine that the test was conducted on has only four CPU cores with two threads on each core, leaving one thread for the Server, another one for the benchmarking user, and the remaining two threads for the background processes to use.

The benchmarking user program runs two asynchronous loops, a sending loop which continuously generates random Input Messages and sends them to the Server, and a receiving loop which waits for Output Messages sent from the Server and calculates the latency and throughput. The sending loop is optimised to generate and send Input Messages with minimal overhead to achieve maximum throughput. Furthermore, to achieve minimum latency, the simulation time delay of the Benchmarking Configuration sent to the Server is set to zero.

From the Queue-based Server benchmark, we recorded maximum throughput values with a different number of concurrent nodes:

- 1 Node: 9.62 Gbps at 256 KB Message Size.
- 2 Nodes: 11.35 Gbps at 256 KB Message Size.
- 3 Nodes: 11.65 Gbps at 512 KB Message Size.
- 4 Nodes: 11.34 Gbps at 512 KB Message Size.

While the maximum throughput values for the Stack-based Server implementation are:

- 1 Node: 9.72 Gbit/s at 256 KB Message Size.
- 2 Nodes: 9.79 Gbit/s at 256 KB Message Size.
- 3 Nodes: 9.94 Gbit/s at 256 KB Message Size.
- 4 Nodes: 9.98 Gbit/s at 256 KB Message Size.

The charts shown in Figure 4.3 visualize the throughput versus the message size for both the Queue-based and Stack-based Server implementations.

We can observe that the Queue-based Server implementation achieves higher throughput values than the Stack-based implementation when more than one node is used. Furthermore, the highest throughput values are recorded between the 256 KB and 512 KB Message Sizes.

We can also observe that with the Queue-based implementation, from a 16 Bytes to a 32 KB Message sizes, the throughput with only 1 node connection is higher than that of multiple node connections. However, with the Stack-based implementation the throughput is approximately the same across the different numbers of node connections.

However, looking at both charts, we can see that with only one node connection, the two implementations give the same results, and the throughput drops to around 6 Gbps after 512 KB and starts rising again after 64 MB message size.

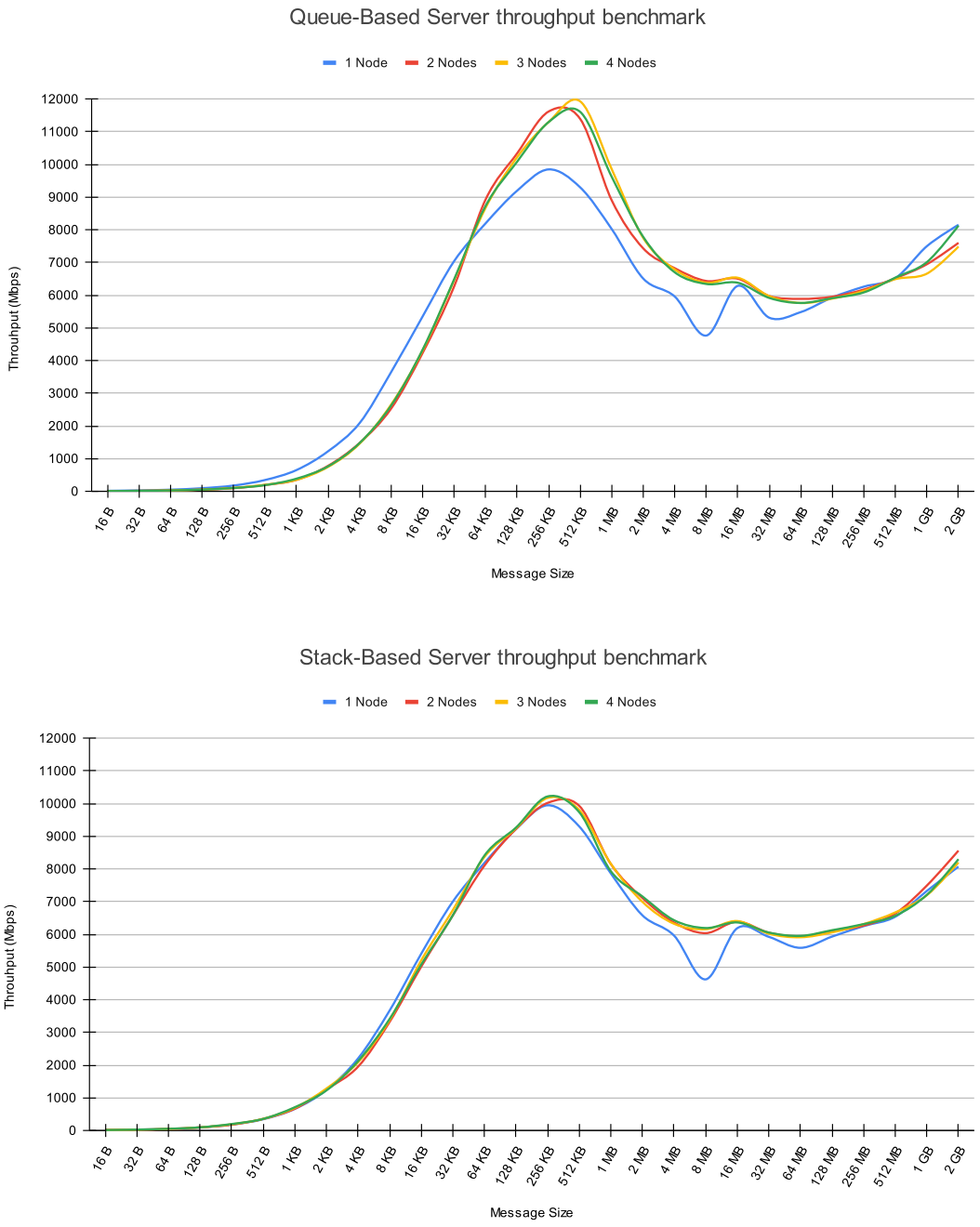


FIGURE 4.3: Graphs of Server throughput benchmark

The charts shown in Figure 4.4 visualize the latency versus the message size for both the Queue-based and Stack-based Server implementations.

From the Queue-based Server benchmark, we recorded minimum latency values with a different number of concurrent nodes:

- 1 Node: 27.68 ms at 256 KB Message Size.
- 2 Nodes: 7.27 ms at 256 KB Message Size.
- 3 Nodes: 6.71 ms at 256 KB Message Size.
- 4 Nodes: 6.71 ms at 256 KB Message Size.

While the minimum latency values for the Stack-based Server implementation are:

- 1 Node: 27.39 ms at 256 KB Message Size.
- 2 Nodes: 29.64 ms at 256 KB Message Size.
- 3 Nodes: 29.12 ms at 256 KB Message Size.
- 4 Nodes: 29.12 ms at 256 KB Message Size.

We can observe that the Queue-based Server implementation achieves lower latency values than the Stack-based implementation when more than one node connection is used, and the lowest latency values are recorded at 256 KB Message size.

Furthermore, the latency behaves the same way as the throughput. i.e. when only one node connection is used, both implementations give approximately the same results. And when using the Stack-based implementation, the different number of concurrent node connections give the same results as the single node connection results. We can also observe that when using the Queue-based Server implementation with more than one node connection, the average latency drops compared to the Stack-based implementation.

From all the benchmarks done on the Server we conclude that using the Queue-based implementation achieves better throughput and lower latency. And the optimal Message Size that should be used to get high throughput and low latency is 256 KB. Hence, the data count in the Input and Output Messages should be calculated for the whole message size to be close to 256 KB.

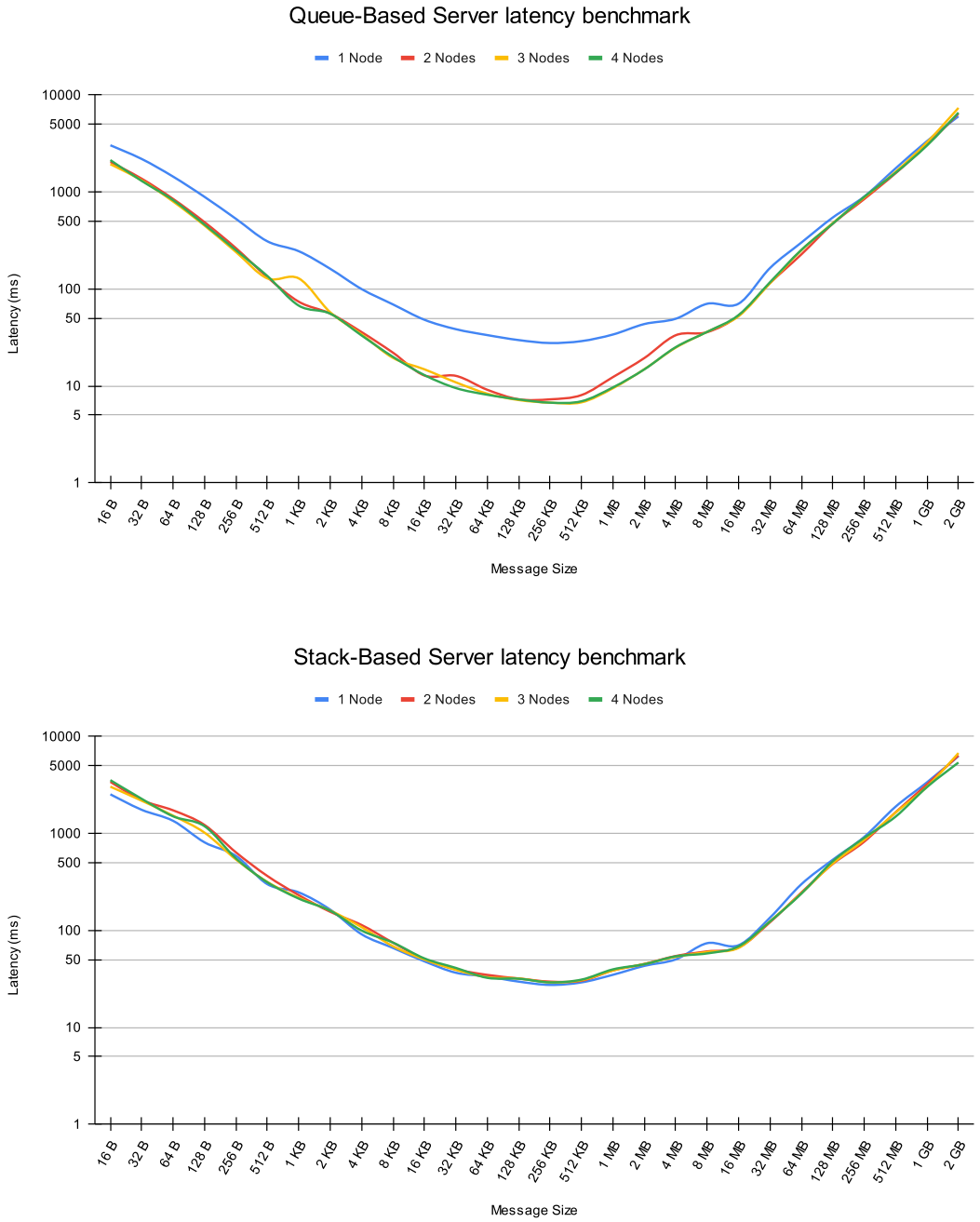


FIGURE 4.4: Graphs of Server latency benchmark

4.2.3 Computing Node Benchmark

To measure the performance of the Computing Node component, we isolated one Node from the system. The isolated node has no computing Hardware Cores which means zero computation time. Following that, we conducted a series of tests to measure the amount of data distributed and collected by the node manager from the Hardware Cores and observed the effect of changing the size of that data on the Node performance and whether a gain was achieved or not. The graph shown in Figure 4.5 is plotted using the recorded results.

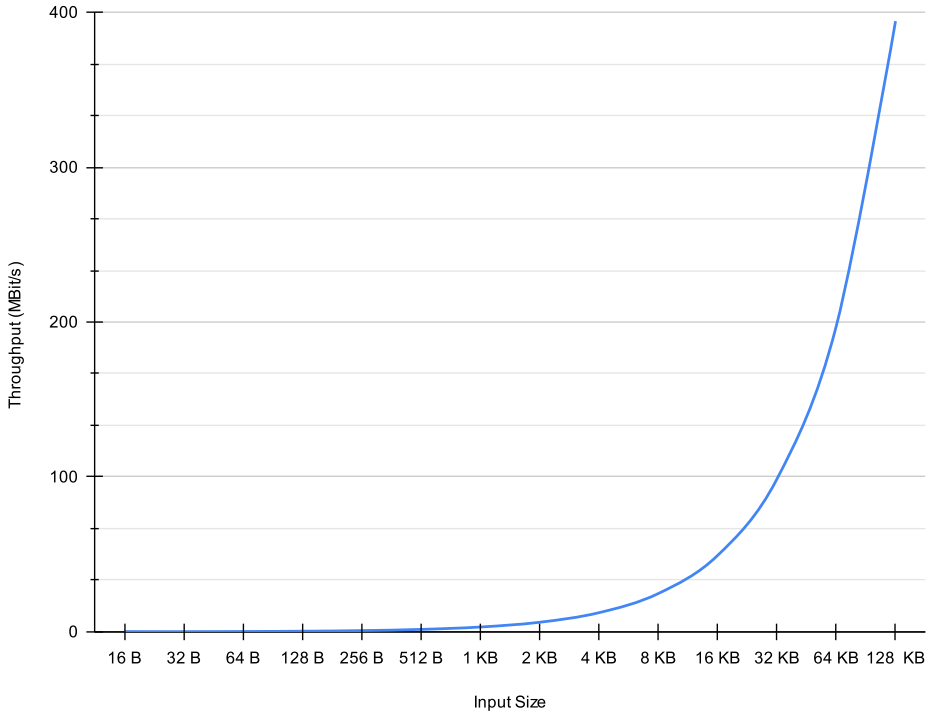


FIGURE 4.5: Graph of load-free Node Throughput Benchmark

The graph of the logarithmic scale of the size of data in bytes versus the Throughput in Mbits/s shows an exponential pattern. This means that the throughput is linearly dependent on the data size. In other words, increasing the data size leads to an increase in the throughput. As observed, 128 kB data size resulted in a throughput of 393 Mbits/s.

These benchmark results can be explained in one logical way. Since the Node manager segments the data into special structures to be allocated, pushed, and popped from and to the queues, smaller data size means a huge number of segments to deal with. On the other hand, the amount of operations to be performed on lager but fewer segments of data is relatively low, this consequences a higher data flow and higher throughput.

Due to the limited memory size inside the node hardware, we couldn't go further with the test using larger data sizes. We believe that the throughput will eventually get closer and closer to the memory reading and writing speed, which is 6 Gbps.

4.3 Test-Case Application

After benchmarking the system on free load conditions, we have designed a specific application to assess the system's performance on real use cases and to compare it with a software-based implementation.

4.3.1 Matrix Multiplier

Multiplying matrices is among the most fundamental and most computationally demanding operations in machine learning and scientific computing[35]. Based on that, matrix multiplication was chosen as an application to test the whole system's performance. In more detail, each core will perform a matrix multiplication on two 64×64 matrices of signed 32-bit integers.

$$(Output)_{64\times64} = (Input1)_{64\times64} \times (Input2)_{64\times64}$$

4.3.2 Matrix Multiplication Hardware Configuration Generation

To create the 64×64 Matrix Multiplier Hardware Configuration, the steps described in section 3.4.1 should be followed. First, a Matrix Multiplier should be designed using HLS. Then duplicates of the Hardware Core are added to the Quartus Template Project that is compiled to generate a Bitsream RBF and the metadata information. Finally, a Hardware Configuration is constructed that can be used by the user to configure nodes to perform distributed Matrix Multiplication.

4.3.2.1 HLS Hardware core

The Matrix Multiplier Hardware Core is implemented with HLS using the naive matrix multiplication algorithm while following the signature described in section 3.4.1.2. The following code snippet shows the HLS component function used for the Hardware Core.

```
#define A_M 64
#define A_N 64

#define B_M 64
#define B_N 64

hls_avalon_agent_component
component void mat_mul_64x64
(
    hls_avalon_agent_memory_argument((A_M*A_N + B_M*B_N) * sizeof(int))
        int* in,
    hls_avalon_agent_memory_argument(A_M*B_N * sizeof(int))
        int* out
    )
{
  for(int i=0; i < A_M * B_N; i++)
    out[i] = 0;

  for(int i=0; i < A_M; i++)
    for(int j=0; j < B_N; j++)
     for(int k=0; k < B_M; k++)
      out[j + i * B_N] += in[k + i * A_N] * in[(A_M*A_N) + j + k * B_N];
}
```

The HLS component is then compiled with the Intel HLS compiler to generate an IP package that is included in the Quartus Template Project to be used in the Intel Platform Designer.

4.3.2.2 Intel Quartus Project Design and Compilation

The generated Matrix Multiplier Hardware Core IP is duplicated 11 times in the Platform Designer and connected to the HPS-to-FPGA bus of the HPS along with the appropriate clock and reset signals. Figure 4.6 shows the Matrix Multiplier Platform Designer system design.

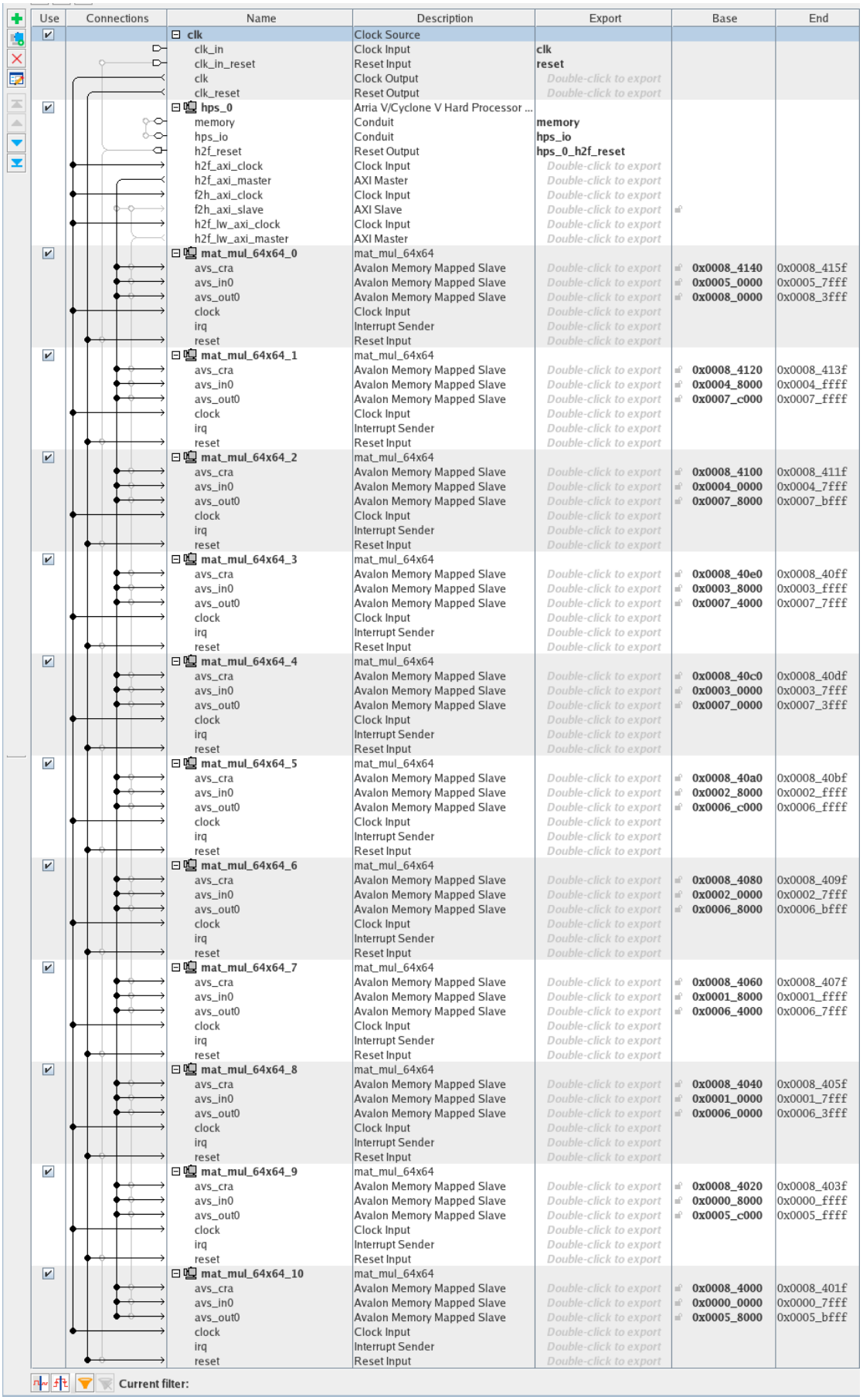


FIGURE 4.6: 64x64 Matrix Multipliers Platform Designer system design

The design is then used to generate the appropriate Verilog files, which are automatically added to the Quartus Template project. After that, the project is compiled to generate an SRAM Object File (SOF) and converted to a Bitsream Raw Binary File (RBF) using the Programming Files Converter available inside the Quartus software.

Figure 4.7 shows the project compilation report generated by Quartus. While Figure 4.8 shows the memory resources utilization summary.

Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	HPS_FPGA
Top-level Entity Name	HPS_FPGA
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	14,504 / 41,910 (35 %)
Total registers	27933
Total pins	338 / 499 (68 %)
Total virtual pins	0
Total block memory bits	4,325,541 / 5,662,720 (76 %)
Total DSP Blocks	22 / 112 (20 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	1 / 4 (25 %)

FIGURE 4.7: Matrix Multiplier Quartus project compilation summary

21	M10K blocks	530 / 553	96 %
22	Total MLAB memory bits	49,500	
23	Total block memory bits	4,325,541 / 5,662,720	76 %
24	Total block memory implementation bits	5,427,200 / 5,662,720	96 %

FIGURE 4.8: Matrix Multiplier Quartus project memory usage summary

From figures 4.7 and 4.8 we can see that 76% of the total memory bits available on the Cyclone V chip are used. However, only 11 multipliers can be fitted in the design because the Quartus Fitter uses M10K blocks for implementing the input and output memory arrays, and 96% of the total M10K blocks available on the Cyclone V chip are used. However, only 35% of the Adaptive Logic Modules (ALMs) are used, which means

more Multipliers can be fitted into the design by optimizing the M10K memory blocks usage or by using the off-chip DRAM memory.

4.3.2.3 Hardware Configuration Construction

Finally, to allow for node configuration from the user of the system, an SMP Configuration Message should be constructed with an Input Size of $64 \times 64 \times 4 \times 2$ which results to 32768 bytes, and an Output Size of $64 \times 64 \times 4$ which results to 16384 bytes. The Number of Hardware Cores is 11 and the Control, Input and Output base addresses of each Hardware Core are obtained from the "Base" column of the Platform Designer system design shown in Figure 4.6. Finally, the generated Bitstream RBF is appended to the end Message, and the total payload size of the SMP Configuration Message is calculated and placed in the Payload Size field. Figure 4.9 shows the part of the SMP Configuration Message of the implemented Matrix Multiplier test-case application.

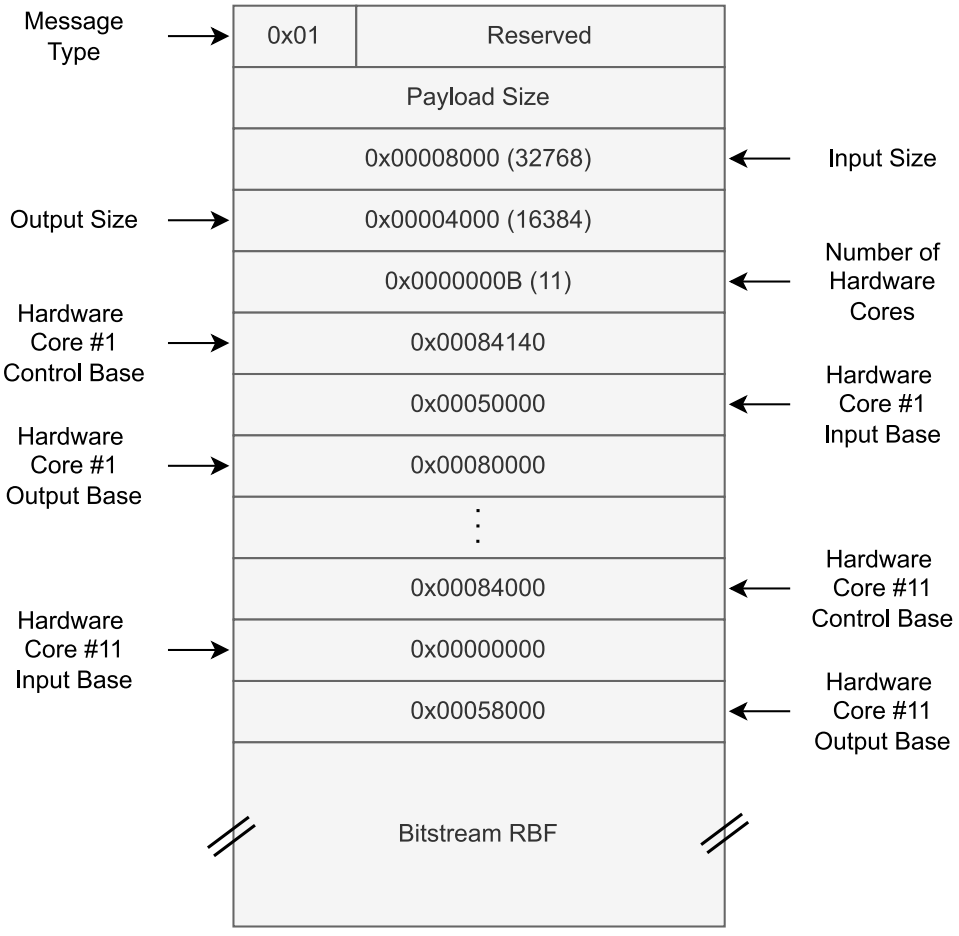


FIGURE 4.9: Matrix Multiplier SMP Configuration Message

4.4 Test Scenario

In order to test the implementation in perfect situations where networking hardware overhead is ignored. First, we configured an isolated node with the Matrix Multiplication Hardware Configuration described in section 4.3.2, and then we collected a set of results where different numbers of cores and data counts were tested, to obtain the optimal points. Next, same as in section 4.2.2, the whole system was simulated on the same machine but instead of using variable time delay in the benchmarking Configuration message, the previously collected results from the Matrix Multiplier Computing Node were used to fix an optimal time delay to simulate a real node.

4.4.1 Configured Node Benchmark

Following the same method in section 4.2.3, and after configuring the node with Matrix Multiplier hardware cores configuration, Random Input matrices were generated inside the node manager and pushed to the computing hardware cores for computation. After the computation is done the result matrix is then read from the core and back to the node manager. From the rate of data flowing in and out of the Hardware cores we could measure the throughput. This process is then repeated ten times while changing the number of working cores each time, and for each core we tested four different data counts. The results of the throughput illustrated in the graph shown in Figure 4.10.

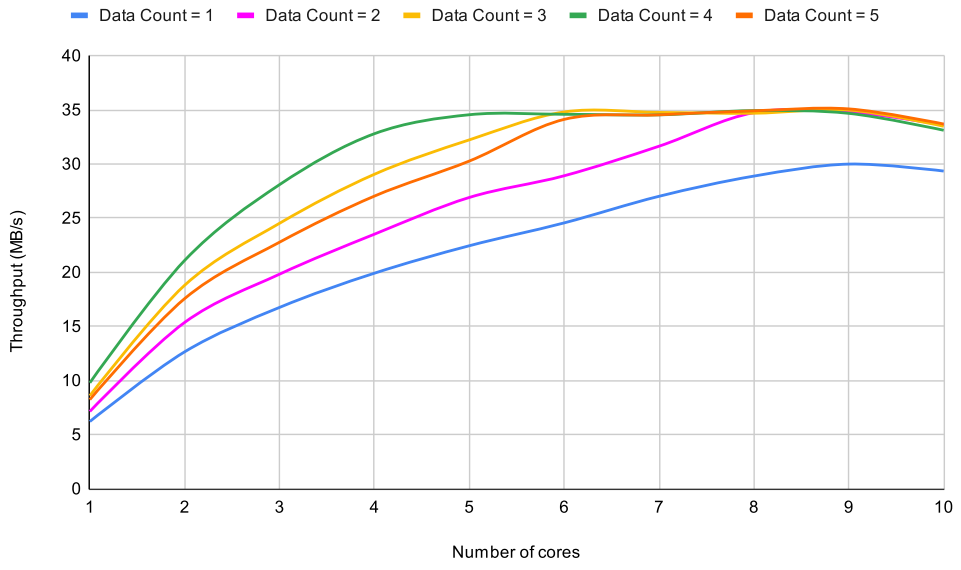


FIGURE 4.10: Graph of Matrix Multiplier Node Benchmark

From the presented data, we can see the effect of incorporating additional Hardware Cores and increasing the data count on the node application performance. For every data

count test, increasing the working hardware cores leads to an increase in the throughput until it reaches its maximum at a certain optimal number of cores, which differs from one data count to another. From the graph, we can determine the optimal number of hardware cores for each data count. The maximum throughput reached was 34 MB/s when the input data count was 2, 3, 4, and 5, while when the input data count was 1, the maximum recorded throughput was 30 MB/s. Noticeably, for 4 data counts, we only need 5 working cores to reach the maximum throughput.

By exploiting the multi-threading concept, we can explain the node's behavior while running multiple thread handlers for the matrix multiplier hardware cores, by saying that an increase in the number of threads that do operations in parallel is beneficial for the performance of the node. On the other hand, there exist a certain number of simultaneous threads that the node processor can handle, and exceeding that number will affect the node negatively because of the overhead of context switching[36]. That appears in the declining pattern of the throughput on the graph, starting from 9 parallel hardware core handlers. Finally, the stable pattern showing in each data count graph starting from a certain number of hardware cores is due to the computation time that the core takes, i.e, there exist always a fixed number of threads working on data exchange while the other threads are always waiting. This is dependent on the core's computation time. Longer computation time means more threads to work on the received data. The computation time for the 64×64 matrix multiplier hardware core was measured to be 7ms per operation.

Considering the achieved results, we can't determine a general optimal number of hardware cores or data count since it's highly dependent on the application, input and output size of the computing hardware core. For this test-case application, the optimal point was 5 simultaneous hardware cores dealing with a data count of 4.

4.4.2 User-Server-Nodes Simulation Benchmark

From the previous section, we saw that a single Computing Node configured with five 64×64 Matrix Multipliers and using an input count of four inputs per Input Message performs around 1054 matrix multiplications per second. Hence, the Computing Node can be seen as a machine that performs matrix multiplication in 949 micro-seconds.
Following the same approach as in section 4.2.2 and using the Benchmarking Hardware Configuration, we setup a Queue-based Server and connected to it different numbers of simulation nodes and a benchmarking user program to simulate the behaviour of the test-case application. However, unlike the benchmarks performed in section 4.2.2 the

simulation time delay is not set to zero but rather set to the processing time taken by the Computing Node to perform a single matrix multiplication.

Three different throughput and latency benchmarking tests were performed.
In the first one, the user sends only one input data per Input Message and Input Messages generation rate is not limited. i.e. the user program generates random Input Messages as fast as possible and sends them to the Server.
In the second test, the Input Messages generation rate is also not limited but the user sends four input data per Input Message.
Finally in the third test, only one input per Input message is sent at a time and the generation rate is fixed. i.e. the number of random Input Messages generated by the user in one second is limited by using a timer between consecutive sends. In this test the simulation time delay used is 1117 micro-seconds according to the results obtained from the configured node application. A Python automation script is written to perform the benchmarks with different numbers of concurrent simulation nodes connected to the Server. And in the third test, multiple user Input Messages generation rates are tested.

The data collected from the first test is shown in table 4.1 The throughput and latency values are plotted in the charts shown in figures 4.11 and 4.12 respectively.

Nodes	Throughput (bit/s)	Avg Latency
1	236.58 MBit/s	500.8 ms
2	464.2 MBit/s	345.5 ms
4	951.84 MBit/s	257.48 ms
8	1.97 GBit/s	205.13 ms
12	3.05 GBit/s	257.85 ms
16	4.09 GBit/s	232.88 ms
20	5.07 GBit/s	218.93 ms
24	5.49 GBit/s	13.76 ms
28	5.56 GBit/s	13.7 ms
32	5.62 GBit/s	13.35 ms
36	5.5 GBit/s	13.62 ms
40	5.21 GBit/s	14.32 ms
42	5.02 GBit/s	14.83 ms

TABLE 4.1: Matrix Multiplier system throughput and latency benchmark with one input count and no user limitation

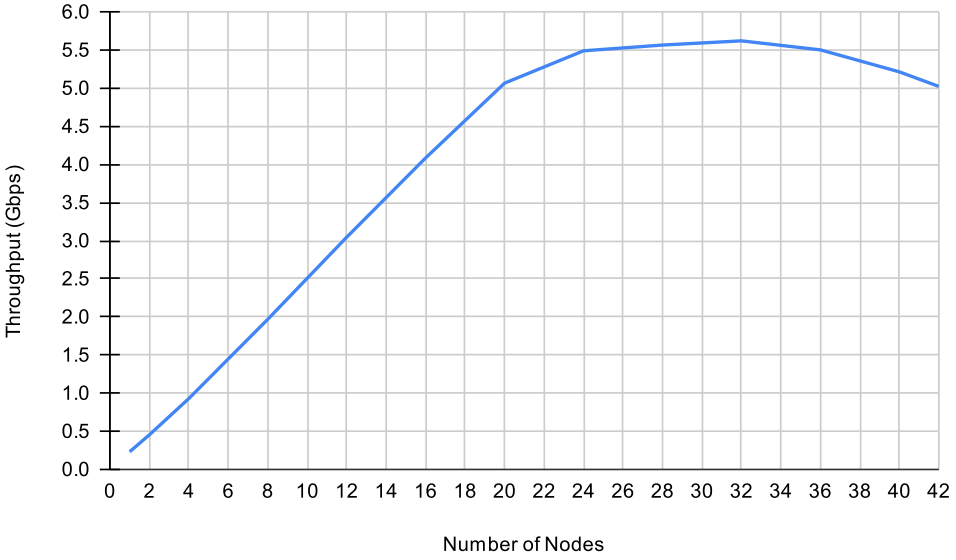


FIGURE 4.11: Matrix Multiplier system throughput benchmark with one input count and no user limitation

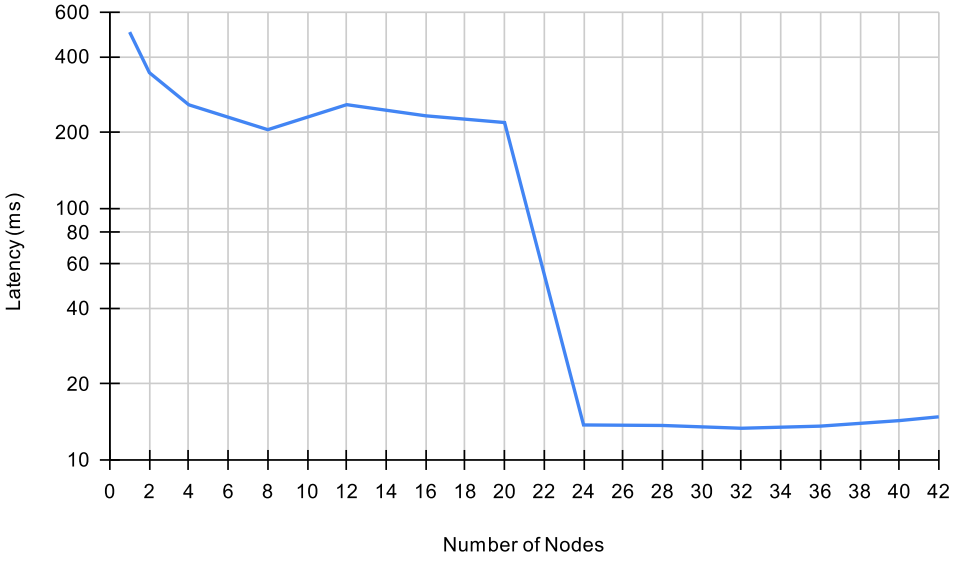


FIGURE 4.12: Matrix Multiplier system latency benchmark with one input and no user limitation

From the first test, we can observe that using only one computing node gives 236.58 Mbps throughput, which is close to the throughput obtained from the isolated node benchmark performed in section 4.4.1. And the average latency is around 500 ms. We can also observe that the throughput increases linearly with every node added. Then the throughput reaches the top limit of 5.62 Gbps after 24 nodes, since in this test we are sending only one input data per Input Message, which means that the SMP Data Message payload size is 32 KB, and from the Queue-based Server Benchmark discussed

in section 4.2.2, we can see that with a 32 KB message size, the maximum throughput is around 6 Gbps.

Furthermore, looking at the average latency, we can see it drop to around 230 ms after more than one connected node, but this is considered relatively high given that the test is performed on the same machine. This is because the Input Messages get stored in the TCP receive buffer of the TCP stack of each node while not getting consumed by the node. We can also observe that at 24 nodes, the latency drops suddenly to around 13 ms. This is because at 24 nodes, the sending throughput of the user is lower than the overall processing throughput of all the connected nodes combined. And this is why using more than 24 nodes does not increase the overall throughput but rather decreases it after 36 nodes due to the Server's routing overhead, while the latency stabilizes at around 13 ms.

Table 4.2 shows the throughput and average latency values versus the number of connected nodes obtained from the second test. The throughput and latency values are plotted in the charts shown in figures 4.13 and 4.14 respectively.

Nodes	Throughput (bit/s)	Avg Latency
1	261.34 MBit/s	539.16 ms
2	519.64 MBit/s	389.5 ms
4	1.01 GBit/s	315.57 ms
8	2.02 GBit/s	282.52 ms
12	3.1 GBit/s	265.66 ms
16	4.13 GBit/s	256.06 ms
20	5.15 GBit/s	226.19 ms
24	6.17 GBit/s	205.91 ms
28	7.13 GBit/s	15.01 ms
32	8.09 GBit/s	12.89 ms
36	8.66 GBit/s	11.96 ms
40	8.34 GBit/s	12.28 ms
44	8.26 GBit/s	12.38 ms

Table 4.2: Matrix Multiplier system throughput and latency benchmark with four input count and no user limitation

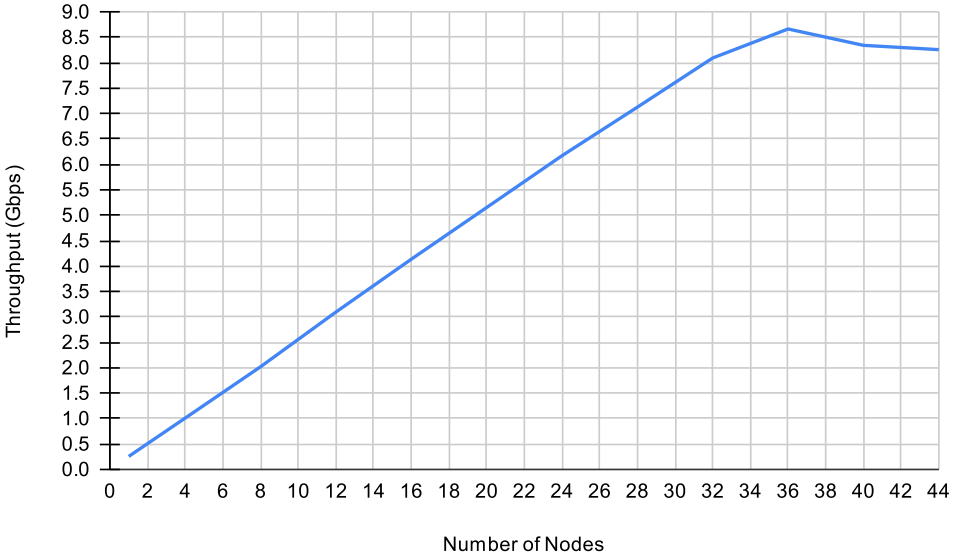


FIGURE 4.13: Matrix Multiplier system throughput benchmark with four inputs and no user limitation

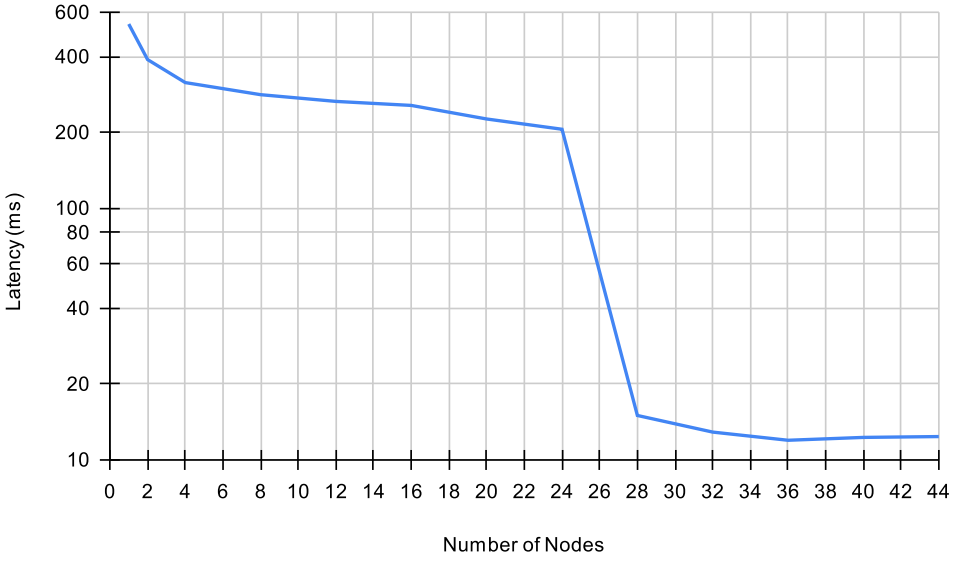


FIGURE 4.14: Matrix Multiplier system latency benchmark with four inputs and no user limitation

From the second test we can observe the same behaviour as in the first test, however, using four inputs per Input Message reached a throughput limit of 8.66 Gbps at 36 connected nodes, this due to the Input Message payload size which in this case is $4 \times 32KB$ which equals to 128 KB, and from the Queue-based Server Benchmark discussed in section 4.2.2, we can see that with a 128 KB message size, the maximum throughput is around 10 Gbps. However, unlike in the first test, this test required 36 computing nodes to reach the maximum throughput. This is because with larger message sizes,

the user's sending throughput increased, which requires more nodes to reach a higher overall computing throughput. Furthermore, we can also see the same behavior in the latency as in the first test, but the latency drop occurred at 28 computing nodes.

The third test is different from the first two tests in that a time delay is added between consecutive Input Messages sent from the benchmarking user to limit the sending frequency. This test is closer to real-world use-cases since most of the applications generate real-time data with fixed frequency or when data is stored on disks, most of which operate at sub-Giga Byte speeds. The test consists of benchmarking the system with different numbers of computing nodes with 1, 2, 5, 10, 50 and 100 Khz Input Messages generation rates.

The throughput and average latency values collected from the third test are shown in tables 4.3 and 4.4 respectively. The throughput data is used to plot the chart of Figure 4.15, and the latency in Figure 4.16.

Nodes	1 Khz	2 Khz	5 Khz	10 Khz	50 Khz	100 khz
1	225 Mbps	247 Mbps	232 Mbps	229 Mbps	228 Mbps	228 Mbps
2	222 Mbps	419 Mbps	466 Mbps	473 Mbps	470 Mbps	456 Mbps
4	222 Mbps	424 Mbps	969 Mbps	962 Mbps	958.0 Mbps	943 Mbps
8	222 Mbps	421 Mbps	967 Mbps	1.7 Gbps	1.99 Gbps	1.95 Gbps
12	220 Mbps	421 Mbps	955 Mbps	1.75 Gbps	3.05 Gbps	2.99 Gbps
16	224 Mbps	421 Mbps	955 Mbps	1.77 Gbps	4.09 Gbps	4.08 Gbps
20	221 Mbps	420 Mbps	986 Mbps	1.78 Gbps	5.11 Gbps	5.09 Gbps
24	221 Mbps	416 Mbps	957 Mbps	1.74 Gbps	5.55 Gbps	5.24 Gbps
28	221 Mbps	422 Mbps	971 Mbps	1.71 Gbps	5.72 Gbps	5.61 Gbps

Table 4.3: Matrix Multiplier system throughput benchmark with one input and limited generation rate

Nodes	1 Khz	2 Khz	5 Khz	10 Khz	50 Khz	100 khz
1	1.42 ms	494.44 ms	461.85 ms	511.06 ms	521.61 ms	517.84 ms
2	1.49 ms	1.47 ms	273.52 ms	334.77 ms	374.08 ms	351.31 ms
4	1.48 ms	1.43 ms	27.91 ms	248.69 ms	254.62 ms	259.24 ms
8	1.5 ms	1.48 ms	1.91 ms	1.34 ms	213.31 ms	207.65 ms
12	1.47 ms	1.47 ms	1.4 ms	1.13 ms	249.46 ms	174.83 ms
16	1.43 ms	1.47 ms	1.27 ms	1.12 ms	230.97 ms	242.53 ms
20	1.5 ms	1.49 ms	1.22 ms	1.12 ms	201.14 ms	220.81 ms
24	1.5 ms	1.51 ms	1.27 ms	1.13 ms	13.16 ms	14.38 ms
28	1.52 ms	1.47 ms	1.24 ms	1.13 ms	8.37 ms	13.63 ms

Table 4.4: Matrix Multiplier system latency benchmark with one input and limited generation rate

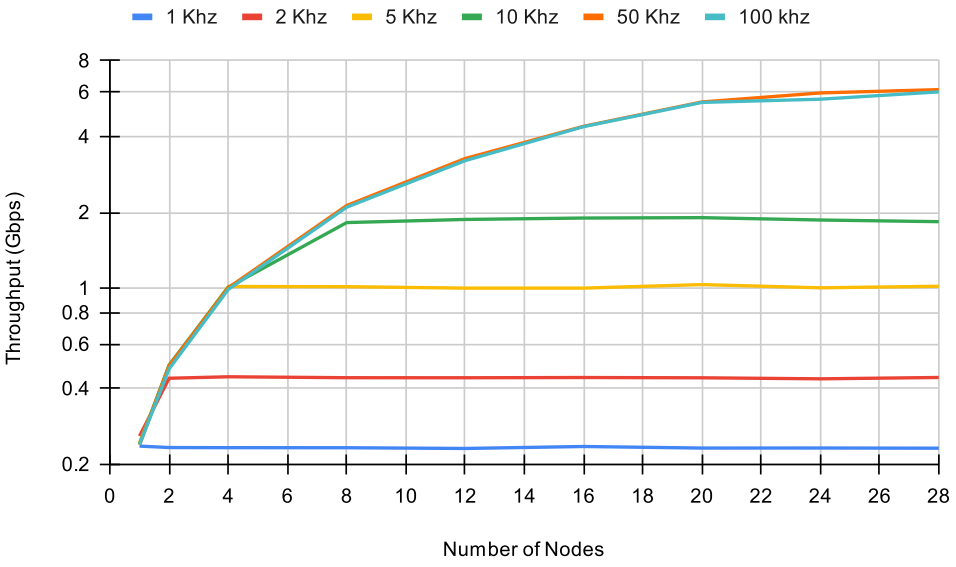


FIGURE 4.15: Matrix Multiplier system throughput benchmark with one input and limited generation rate

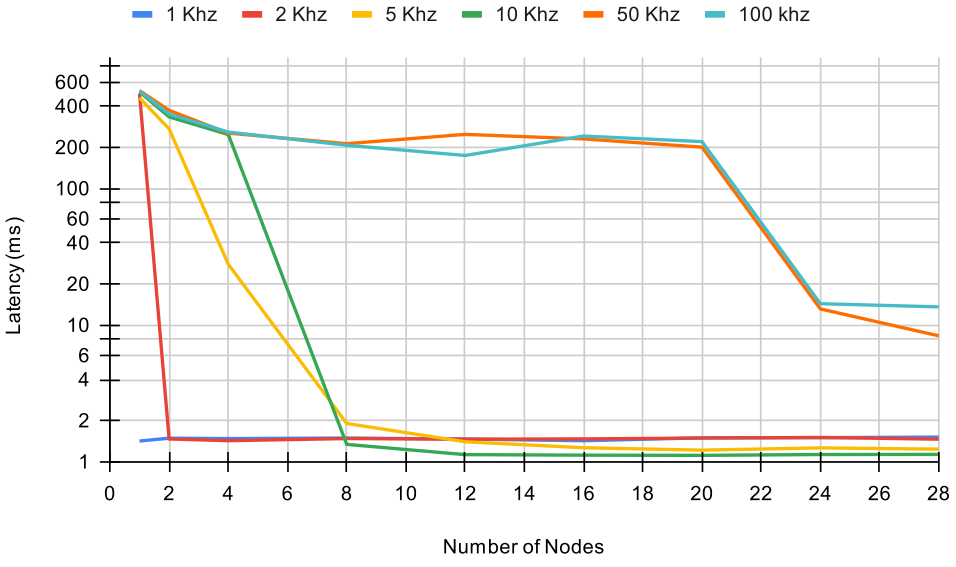


FIGURE 4.16: Matrix Multiplier system latency benchmark with one input and limited generation rate

From this test, we can observe that each frequency reaches its full processing throughput depending on the number of connected nodes. The 1, 2, 5, and 10 Khz frequencies require 1, 2, 4, and 8 computing nodes, respectively, so that the processing throughput matches the user throughput. However, for the 50 Khz and 100 Khz frequencies, the throughput increases for every node added until they reach the maximum Server throughput of around 5.7 Gbps.

Furthermore, by observing the latency chart, we can see that the latency drops to around 1.4 ms when the processing throughput matches the user throughput. And for the 50 Khz and 100 Khz frequencies the latency does drop to a lower value but stays relatively high. And since the matrix multiplication takes around 1117 ms to complete, we can conclude that when the overall processing power matches the user's throughput, the server's overhead is around 283 micro-seconds.

4.5 Software-Based Implementation

To evaluate the performance of the system under its optimal conditions, a software-only implementation of 64×64 signed 32-bit integer matrix multiplier was written in C++ for comparison. For fair comparison, the function used in the software application is similar to the one used in the HLS code. The output of the C++ code compiled on full optimization produced a Throughput of 96 MB/s.

4.6 Preliminary Conclusions

Looking over the data collected for all the various system configurations, we can draw some preliminary conclusions. First, changing multiple variables and configurations in the system, namely payload size, number of boards, and number of hardware cores, demonstrated an improvement in system performance. Second, the optimal system configuration depends on the application itself, and a general formula can't be obtained only by testing the use case application. Finally, the system, configured with a test-case application, showed better results than a software implementation of the same application even without using an optimized hardware design.

- Using 4 nodes : $\times$1.3 faster
- Using 8 nodes : $\times$2.6 faster
- Using 16 nodes : $\times$5.5 faster
- Using 32 nodes : $\times$10.7 faster
- Using 36 nodes : $\times$11.5 faster

Chapter 5

Conclusion and Future Work

This work involved designing and implementing a distributed system that provides users with a framework to configure and interact with multiple FPGA boards. We showed how flexible the system is and its ability to adapt to various application requirements by accepting variable input and output sizes. We have also proven that the system is scalable in that it could support multiple computing hardware cores and FPGA boards. After separating the system into components, we performed an analysis on each component to observe the system's overhead on the user application. Using a test-case application, the system was benchmarked on its optimal configuration (data count, number of cores). As a result, the system behavior followed a logical pattern, incorporating more computing nodes led to an increase in the general throughput and a decrease in the latency. Finlay, the evaluation of the system was based on a comparison with a software-based implementation of the same test application, in which the system performed better with factors of 1.3, 5.5, and 11.5 using 4, 16, and 36 nodes, respectively.

In the future, our main focus will be on extending the networking stack by designing a custom transport protocol with reliability and flow control functionalities, this will best suit our implementation instead of using the TCP protocol. In addition, we will also extend the SMP protocol to provide support for different FPGA chips from different vendors, and add real-time system performance reporting to better manage the computing nodes. For the server part, we are planning to implement a multi-threaded architecture rather than the current single threaded implementation. Finally, instead of using development boards as the computing nodes, we will use a motherboard with more powerful hardware and multiple FPGA boards connected to it through the PCIe interface to achieve better performance and further parallelize the system at the computing node level.

Bibliography

[1] G.E. Moore. "Cramming More Components Onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85. DOI: 10.1109/JPROC.1998.658762.

[2] R.H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[3] Hadi Esmaeilzadeh et al. "Dark silicon and the end of multicore scaling". In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376.

[4] Jason Cong. "A new generation of C-base synthesis tool and domain-specific computing". In: *2008 IEEE International SOC Conference*. 2008, pp. 386–386. DOI: 10.1109/SOCC.2008.4641556.

[5] *White Paper: Vivado Design Suite*. URL: https://docs.xilinx.com/v/u/en-US/wp416-Vivado-Design-Suite.

[6] Tomasz S. Czajkowski et al. "From opencl to high-performance hardware on FPGAS". In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 2012, pp. 531–534. DOI: 10.1109/FPL.2012.6339272.

[7] Morgan Kaufmann. *Computer Networking: A Top-Down Approach*. S. Hauck and A. DeHon, 2007.

[8] Peter M. Shirazi Nabeeland Athanas and A. Lynn Abbott. "Implementation of a 2-D fast Fourier transform on an FPGA-based custom computing machine". In: *Field-Programmable Logic and Applications*. Ed. by Will Moore and Wayne Luk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 282–292. ISBN: 978-3-540-44786-3.

[9] Russell J. Petersen and Brad L. Hutchings. "An assessment of the suitability of FPGA-based systems for use in digital signal processing". In: *Field-Programmable Logic and Applications*. Ed. by Will Moore and Wayne Luk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 293–302. ISBN: 978-3-540-44786-3.

[10] Abdullah Alkalbani, Teddy Mantoro, and Abu Osman. "Comparison between RSA hardware and software implementation for WSNs security schemes". In: Jan. 2011, E84–E89. DOI: 10.1109/ICT4M.2010.5971920.

[11] *Digital System Design with FPGA: Implementation Using Verilog and VHDL 1st Edition*. McGraw Hill 1st edition, July 10, 2017.

[12] Intel. "Cyclone V Hard Processor System Technical Reference Manual". In: (8 July 2021).

[13] ARM. "Cortex-A9 MPCore Technical Reference Manual r4p1". In: (June 2012).

[14] ARM. "AMBA AXI and ACE Protocol Specification". In: (26 Jan 2021).

[15] *The Linux Kernel documentation*. URL: https://docs.kernel.org/.

[16] Intel. "Intel High Level Synthesis Compiler Pro Edition Best Practices Guide". In: (20 June 2022).

[17] Intel. "Platform Designer User Guide". In: (29 March 2022).

[18] Terasic. "DE-10 Standard User Manual". In: (20 March 2012).

[19] *Terasic DE10-Standard Development Kit*. January 2019. URL: https://www.rocketboards.org/foswiki/Documentation/DE10Standard.

[20] *Cyclone® V FPGAs and SoC FPGAs*. URL: https://www.intel.com/content/www/us/en/products/details/fpga/cyclone/v.html.

[21] packt. *Cyclone V SOC-FPGA Overview*. December 2018. URL: https://subscription.packtpub.com/book/iot-&-hardware/9781788629300/11/ch11lvl1sec92/hybrid-fpgasoc-chips.

[22] *M10K memory block definition*. URL: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_m10k.htm.

[23] *Adaptive Logic Module (ALM) definition*. URL: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_alm.htm.

[24] Intel. *Address maps of Cyclon V interconnect*. February 20222. URL: https://www.intel.com/content/www/us/en/products/details/fpga/cyclone/v/article.html.

[25] *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Pearson 7th edition, April 26, 2016.

[26] Kaveh Pahlavan and Prashant Krishnamurthy. "IEEE 802.3 Ethernet". In: *Networking Fundamentals: Wide, Local and Personal Area Communications*. 2009, pp. 343–388. DOI: 10.1002/9780470779422.ch8.

[27] Jon Postel. "Internet Protocol". In: *DARPA Internet Program Protocol Specification* RFC 791 (1981).

[28] Ralph Droms. "Dynamic Host Configuration Protocol". In: RFC 2131 (March 1997).

[29] Jon Postel. "Transmission Control Protocol". In: *DARPA Internet Program Protocol Specification* RFC 793 (1981).

[30] et al. Allman. "TCP Congestion Control". In: RFC 5681 (2009).

[31] *Round-robin scheduling*. URL: https://en.wikipedia.org/wiki/Round-robin_scheduling.

[32] Robin Sebastian. *Rsyocto*. 2019. URL: https://github.com/robseb/rsyocto.

[33] *Throughput*. September 2022. URL: https://www.techtarget.com/searchnetworking/definition/throughput.

[34] Cloudfare. *Latency*. September 2022. URL: https://www.cloudflare.com/learning/performance/glossary/what-is-latency/.

[35] Davis Blalock and John Guttag. "Multiplying Matrices Without Multiplying". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 992–1004. URL: https://proceedings.mlr.press/v139/blalock21a.html.

[36] Dov Bulka and David Mayhew. "Efficient C++: performance programming techniques". In: *Kybernetes: The International Journal of Systems Cybernetics* 29 (Jan. 2000).