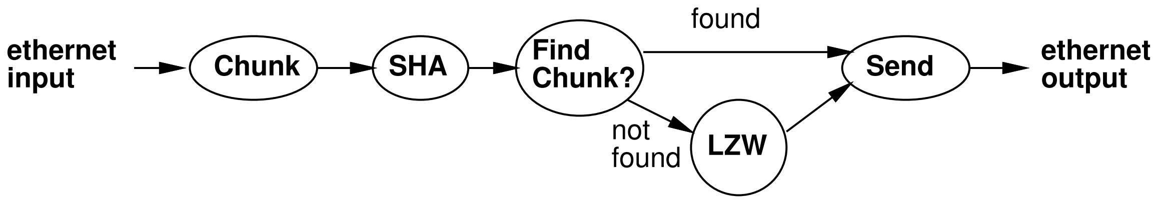
University of Pennsylvania
Department of Electrical and System Engineering
System-on-a-Chip Architecture

ESE5320, Fall 2023 Deduplication and Compression Project Wednesday, October 25

Due: Monday, December 11, 10:15AM (demo), 5:00PM (writeup)

1 Goal

Develop a compressor that can receive data in real time at modern ethernet speeds and compress it into memory using deduplication and compression. Specifically, we'll look at Content-Defined Chunking to break the input into chunks, SHA-256 (or SHA3-384) hashes to screen for duplicate chunks, and LZW compression to compress non-duplicate chunks. For full points, your goal for implementation is to achieve real-time guaranteed support of 400 Mb/s[1] but you may need to consider intermediate goals (e.g. 100Mb/s, 200Mb/s) along the way. Slower designs will receive partial credit for the performance portion of the project grade.



This is a 5-week project assignment; the intent is to allow you to plan and execute a significant, open-ended design exploration and mapping. You will not achieve the implementation goal or the course learning goals by trying to do this in one week. We give you milestones to help provide some structure, but the milestones are minimal and *doing the minimum to hit the milestone each week will be insufficient* to get you where you need to be at the end. We are giving you flexibility in planning and ordering rather than lock-step specifying exactly what you need to do each week.

- Project work is done in teams of 3. You select partners during first week.
- Collaboration between teams is limited as specified on the course web page.
- Milestones writeups and Final report are a team writeup.
- The spirit of this exercise is to optimize the SoC mapping of the algorithm. As such, explorations of alternate solutions that change the algorithms and generally optimize the solution for hardware and software are out of scope. Explorations that tweak or tune the algorithms slightly to better exploit the SoC hardware are potentially in scope.

[1] 800 Mb/s stretch goal for bonus points

2 Final Report

Final report is a team writeup. There will be one turnin per team.

- Describe your single ARM processor mapped design. [1 page]
  - Key parameters in the solution
  - Performance achieved (also include associated `-s` parameter to client to demonstrate)
  - Compression achieved
  - Characterization and breakdown of time spent in the major components
- Describe your final Ultra96 mapped design. [5 pages]
  - Performance achieved and energy required
  - Compression achieved
  - Key design aspects: task decomposition, parallelism, mapping to Zynq resources, include diagrams to support
  - Be clear where each component of the final design is performed (e.g., ARM, NEON vector, FPGA logic).
  - Model to explain performance.
  - Current bottleneck preventing higher performance
- Describe how you validated your implementations, including the real-time guarantee for the input rate. The real-time validation likely includes both arguments about the way the code is written and mapped to the Zynq and your testing methodology. [2 pages]
- Describe the key lessons you learned from this design experience. [1 page]
- Describe design space explored and show graphs and models to support design selection. [any number of pages as needed]
- Describe who did what. [1 page]
- Include academic integrity statement for all team members:

> I, *your-name-here*, certify that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this final exercise.

You can review the Code of Academic Integrity here: https://catalog.upenn.edu/pennbook/code-of-academic-integrity/

3 Final Project Code and Bitstream Submission

We intend to run your compression routines. To make sure the process is consistent across teams, please comply with the following standards.

1. Provide an xclbin, OpenCL host code executable, and decoder executable for your encoder.
   - Turn in a tar file to the designated final implementation assignment on canvas.
   - One turnin for team.
   - Should be a single tar file, containing seven files:
     - `encoder.xclbin, BOOT.bin, boot.scr, image.ub` for FPGA kernel
     - `encoder` for OpenCL host code executable
     - `decoder` executable configured to work with your encoded file. (Most likely, this is just a compilation of the `Decoder.cpp` we supplied; however, if you chose a different maximum block size, you may need to change `CODE_LENGTH`; so give us back one with that change made.)
     - `client.sh` shell script to invoke client with suitable `-s` parameter to demonstrate your guaranteed data transfer performance.
2. Your compression program (OpenCL host code) should take one argument:
   - the file name where the program should store the compressed data.

   Your program should assume that `encoder.xclbin` is in the same directory as the host executable.
3. Your compression program should start up ready to receive inputs.

We may provide further clarification or revision on this final implementation turnin as we get closer.

3.1 Demo Day

We will use the final lecture slot on Monday, Dec. 11th for you to demonstrate your encoder to course staff. Each team will singup for a slot during the lecture period. Signup will be made available on Ed Discuss between now and Dec. 11th.

4 Milestones

We will provide precise requirements for milestones each week. These may include a few exercises to help prepare you for questions that may be on the final in addition to the project specific components. Milestones and feedback feed into the final report. In most cases, the milestones can serve as a first draft of a component of your report, and the feedback we give you will help provide guidance on how to refine it for the report.

1. Analysis, parallelism, placeholder encoder, and teaming [11/3]
2. Functional version and design space [11/10]
3. First operator on FPGA [11/17]
4. Intermediate throughput design (e.g., try for portion working at 100–200 Mb/s) [12/1]
5. Final Report [12/11]

5 Components

The components we will use are standard enough that the wikipedia pages are useful, and there are several other nice tutorial blog posts out there. Here's a roundup of starting points.

- Content-Defined Chunking (Rabin Fingerprint)
  - `https://moinakg.wordpress.com/tag/rabin-fingerprint/`
  - `https://restic.net/blog/2015-09-12/restic-foundation1-cdc`
  - `https://en.wikipedia.org/wiki/Rabin_fingerprint`
- SHA 256 (SHA3-384) Hashing
  - `https://tools.ietf.org/html/rfc6234`
  - `https://en.wikipedia.org/wiki/SHA-2`
  - `https://en.wikipedia.org/wiki/SHA-3`
  - An example possibly useful for testing: `https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/SHA256.pdf`
  - More examples (including some for SHA3-384) can be found `https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values`
  - You may use the hardwired SHA unit on the ZCU3EG.
    - ∗ We provide you a platform that enables the SHA unit in the course directory on eniac: `/home1/e/ese5320/u96v2_sbc_crypto_base`
      - · Sample code for using: `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841654/Linux+SHA+Driver+for+Zynq+Ultrascale+MPSoC`

      · *N.B.* To use this unit, data provided must be padded to a multiple of 4B to get consistent results; providing this padding and extracting full performance can be tricky.
  - You may use a software implementation that exploits NEON intrinsics.
    - https://github.com/james-ben/mpsoc-crypto
    - https://github.com/noloader/SHA-Intrinsics
- Lempel-Ziv-Welch Compression
  - http://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/
    - This is a good overview of the algorithm. However, you should use this for the pseudocode to outline an approach. The C++ code they provide uses datatypes that **will not work** for your application where you must work on binary data. You also need to think about how you implement the map.
  - https://en.wikipedia.org/wiki/Lempel-Ziv-Welch

6 Some Suggested Parameters

There will be some discretion in picking implementation parameters. From the start we will suggest considering:

- 4KB average chunk size with a 8KB maximum
- Use on-board DRAM for the chunk dictionary and hash fingerprints
- Full, maximum chunk size as the LZW compression window

You may want to experiment with some of the parameters when tuning your implementation.

7 Examples of Use

1. Yan Zhang, Nirwan Ansari, Mingquan Wu, and Heather Yu. "On Wide Area Network Optimization." In *IEEE Communications Surveys & Tutorials*, vol. 4, issue 4, pp. 1090–113, Oct. 2013. http://ieeexplore.ieee.org/document/6042388/ Sections III A and B survey the role of compresison and decompression in optimizing WAN data traffic.
2. Ashok Anand, Chitra Muthukrishnan, Aditya Akella, and Ramachandran Ramjee. "Redundancy in Network Traffic: Findings and Implications". In *Proceedings of ACM SIGMETRICS/Performance*, 2009. https://www.microsoft.com/en-us/research/publication/redundancy-in-network-traffic-findings-and-implications/ Characterizes redundancy in network traffic.
3. Athicha Muthitacharoen, Benjie Chen, and David Mazieres. 2001. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP '01)*. pp. 174-187. https://dl.acm.org/citation.cfm?id=502052 Use of deduplication for optimizing a file system operating across a low bandwith link.

8 Other Resources

- Xilinx HLS Tiny Tutorials https://github.com/Xilinx/HLS-Tiny-Tutorials. These show examples of how to write code for specific things in Vitis HLS.
- Vitis Tutorials https://xilinx.github.io/Vitis-Tutorials/master/docs/README.html. These are written for data center platforms (AWS F1/Alveo cards), but the decomposition and core routines should still be useful.
- For examples of other applications that have been converted to run with PL accelerators in HLS, you can look at:
  - the Rosetta Stone Benchmarks https://github.com/cornell-zhang/rosetta from Cornell. These were written for SDSoC, but the decomposition and core routines should still be useful.
  - Parallel Programming for FPGAs by Kastner et al. http://kastner.ucsd.edu/wp-content/uploads/2018/03/admin/pp4fpgas.pdf

9 Encoded Data Storage

For your timing runs, store the compressed data in DRAM. The largest test case we provide is under 200MB. Copy your encoded data from DRAM to the SD-Card outside of your test timing. You will need to plan how you divide the DRAM among buffers, chunk hash storage, and compressed output. During early testing, before adding external input, you may also want to start with the uncompressed data in DRAM.

10 Ethernet Packet Block Size

You will transfer data in ethernet packets. Your compression should be **invariant** to the size of the ethernet packet. That is, where ethernet packets end and begin should not impact where you choose for chunks to begin or end. *Chunks will typically span across packet boundaries.* Ending chunks at packet boundaries will harm deduplication since the packet sizes are unrelated to the content that you are trying to identify and deduplicate.

11 Chunk Validation

Using an SHA-256 signature, the probability of having a collision where two chunks share the same signature is extremely low. For the project, we will consider equality of SHA-256 signatures adequate to determine that a chunk is a duplicate. This means you do not need to read back the chunk and validate that it is, in fact, identical. If you had terabytes of data, or if the consequences of error were high, you would want to perform the check. This only applies to the full 256b signature. If you use smaller hashes for indexing, you will still need to validate that there is a match on the 256b signature.

12 Compressed Format

- Compressed stream is a sequential concatenation of chunks.
- Each chunk has a 32b header that identifies it as Duplicate Chunk or LZW Chunk.
  - A Duplicate Chunk is a 32b value
    - bit 0 is a 1 to signify a Duplicate Chunk
    - bits 31–1 is the Chunk Index of previously encoded block to be duplicated. Only LZW Chunks are indexed. The first LZW chunk has index 0, the next 1, etc.
  - An LZW Chunk is
    - a 32b bit header
      - bit 0 is 0 to signify an LZW Chunk
      - bits 31–1 is the *compressed* chunk length in bytes
    - LZW-compressed contents of the chunk. LZW implementations vary. Our implementation satisfies the following properties:
      - Entries 0–255 of the dictionary are initialized to the 256 literals, e.g. a byte with value 27 would be encoded as 27.
      - The next dictionary entries are used for prefixes: sequences of 2 or more bytes. The dictionary is expanded on every code word sent (received). The next definition is the dictionary entry for the decoded string from the previous code word extended by the next byte following the decoded code word string in the plain text. This follows the standard LZW encoding as reviewed in: http://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/.
      - No special keywords such as end-of-file are contained in the dictionary.
      - The dictionary size is only limited by the chunk size limitation.
      - For simplicity, we set all code lengths to be $\lceil \log_2 MaxChunkSize \rceil$ in this project. The provided *decoder* works with the *MaxChunkSize = 8192 bytes*. You can change the $MaxChunkSize$ by changing the defined parameter `CODE_LENGTH`.
      - Code words are output MSB-first. Assuming nothing has been output yet, a 13b code with binary value $x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0$ results in two consecutive bytes with values $x_{12}x_{11}x_{10}x_9x_8x_7x_6x_5$ and $x_4x_3x_2x_1x_0000$. The next code word with value $y_{12}y_{11}y_{10}y_9y_8y_7y_6y_5y_4y_3y_2y_1y_0$ changes the second byte to, $x_4x_3x_2x_1x_0y_{12}y_{11}y_{10}$, the third to $y_9y_8y_7y_6y_5y_4y_3y_2$ and the fourth to $y_1y_0000000$.
      - *N.B.*, You will need to pay attention to endianess in your hardware implementation to make sure it is consistent with the decoder.
    - Padding so that the entire LZW chunk ends on an 8b boundary; that is, chunks of either type always begin on 8b boundaries.

Because we are using a uniform length of $\lceil \log_2 MaxChunkSize \rceil > 8$ for encoding codewords, it is possible that an encoded chunk could be longer than the unencoded chunk. To deal with this, real implementations will often compare the length of the LZW encoded chunk to the raw chunk length and send the unencoded data if it is smaller. For simplicity, we are not asking you to perform that optimization (and our encoded chunk format does not support it).

13 Compression Goal

Changes in parameters (such as average chunk size) will change deduplication and compression results. Furthermore, you may make tradeoffs in implementation that impact compression ratios. You may choose to sacrifice some level of compression for throughput. You should try to maximize deduplication and tradeoff only a modest (e.g. 10%) level of chunk compression. Show your tradeoffs explored in your detail design-space exploration section with graphs to support as apporpriate.

14 Supplied Resources

- Laptop code and/or scripts to send data to your Ultra96 at a fixed (tunable) frequency (Section 15).
- Dummy design to receive packets from sender (Section 15).
- Reference implementation of decoder (see `project/Decoder` in course code repository).
  - Your encoder needs to work with the decoder. Getting the encoder to produce encodings according to the compressed format (Sec. 12) may take some experimentation. You may want to add a verbose debugging option to the decoder to have it print out what values it is getting for the various fields to expedite the debugging of your encoder.
- We provide several datasets that you can use for testing. We encourage you to create your own simple datasets for unit testing. Note that the tar-files are not meant to be unpacked. Following are the datasets that we provide.
  - The Little Prince unencoded. As an encoding example, we also provide The Little Prince compressed.
  - Simple example. This archive contains three files, two of which are identical.
  - Benjamin Franklin's autobiography. This is a simple text file that you can modify for your own purposes. The current file probably has few duplicate areas. (390 KB)
  - GTK+ source code. This file contains several subsequent versions of the GTK+ source, which provides ample opportunity for deduplication. (177 MB)
  - Linux source code. This file contains several subsequent versions of the source. (191 MB)
  - Several Linux kernels. As opposed to the other data sets, this set contains prevalently binary data. (66 MB)

  Note: you should take these as examples, not a definitive list of test cases. In particular, *you should create many other focused test examples to facilitate your debugging and validation.*

15 Ethernet Setup and Supplied Code

15.1 Measure Ethernet Speed

1. Make sure that you have the following pre-requisites figured out (you should already have this setup from homework 4):

   (a) Communication over ethernet. If you are using Windows, follow this document. If you are on Mac, use the following instructions:

      i. Download and install the AX88179 driver in the Mac (which is for the ethernet usb):

      ii. And then in Mac, you can do `screen /dev/tty.usbserial-1234_oj11 115200` to open the serial console for the Ultra96. Assign the ip address to ultra96 like you would do normally (see Homework 6 instructions). You can exit the serial console by doing `CTRL-A CTRL-\` and pressing `y`.

      iii. Once the ethernet driver is installed on your Mac, you can assign it an ip address using `sudo ifconfig en4 10.10.7.2 netmask 255.0.0.0` where `en4` is the interface name you will find from ifconfig.

   If you are using a Linux machine in Detkin/Ketterer as your host machine, use the command: `ifconfig_eth1`. Note that if that command didn't work, you might have to use `ifconfig_eth2` or `ifconfig_eth3`. These commands are equivalent to the command:

   ```
   sudo ifconfig ethX 10.10.7.2 netmask 255.0.0.0
   ```

   This is the only way to assign an IP to the USB-ethernet device in the Linux machines in Detkin and Ketterer.

   (b) Install `iperf3` on your computer: https://iperf.fr/iperf-download.php

   (c) Find out what kind of USB ports you have in your computer: USB-2.0 or USB-3.0.

2. Open the Ultra96 terminal and issue the command: `/usr/bin/iperf3 -s`

3. Open a terminal in your computer and issue the command: `/usr/bin/iperf3 -c 10.10.7.1` (assuming `10.10.7.1` is the IP address you assigned to the Ultra96).

4. You should see outputs similar to the following if you connected to a USB-3.0 port in your computer:

```
Connecting to host 10.10.7.1, port 5201
[  5] local 10.10.7.2 port 38484 connected to 10.10.7.1 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   110 MBytes   920 Mbits/sec    0    266 KBytes
[  5]   1.00-2.00   sec   109 MBytes   917 Mbits/sec    0    266 KBytes
[  5]   2.00-3.00   sec   109 MBytes   916 Mbits/sec    0    276 KBytes
```

```
[  5]   3.00-4.00   sec   109 MBytes   915 Mbits/sec    0    276 KBytes
[  5]   4.00-5.00   sec   106 MBytes   893 Mbits/sec    0    287 KBytes
[  5]   5.00-6.00   sec   104 MBytes   874 Mbits/sec    0    287 KBytes
[  5]   6.00-7.00   sec   105 MBytes   878 Mbits/sec    0    287 KBytes
[  5]   7.00-8.00   sec   105 MBytes   877 Mbits/sec    0    287 KBytes
[  5]   8.00-9.00   sec   105 MBytes   880 Mbits/sec    0    287 KBytes
[  5]   9.00-10.00  sec   105 MBytes   878 Mbits/sec    0    287 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  1.04 GBytes   895 Mbits/sec    0             sender
[  5]   0.00-10.00  sec  1.04 GBytes   893 Mbits/sec                  receiver
iperf Done.
```

This tells you that the upper bound on the throughput achieved by a placeholder receiver is around 895 Mb/s limited by the Ugreen ethernet-to-USB interfaces.

15.2 Obtaining Starter Code and Integrating Ethernet Input

We will now describe how data will be sent through ethernet using the client and server, the packet layout, and other relevant information for getting started to receiving real-time data.

1. Clone the `ese532_code` repository using the following command:

   ```
   git clone https://github.com/icgrp/ese532_code.git
   ```

   If you already have it cloned, pull in the latest changes using:

   ```
   cd ese532_code/
   git pull origin master
   ```

   The code you will use for this section is in the `project` directory. The directory structure looks like this:

   ```
   project/
     Client/
       client.cpp
     Decoder/
       Decoder.cpp
     Server/
       encoder.cpp
       encoder.h
   ```

```
          event_timer.cpp
          event_timer.h
          server.cpp
          server.h
        sourceMe.sh
        LittlePrince.txt
        Makefile
```

2. If you are compiling on BigLab/Detkin/Ketterer, run `sourceMe.sh`. If you are building locally in your own machine, make sure to source Vitis settings, e.g.:

   ```
   source /opt/Xilinx/Vitis/2020.2/settings64.sh
   ```

   And make sure the `PLATFORM_REPO_PATHS` is setup to the platform you downloaded.

3. You can either use make or the Vitis GUI to compile your code. Use `make all` to compile all the targets `client`, `encoder`, and `decoder`.

4. Use `make clean` to clean all the generated files.

5. Our basic model will be communication between two systems—your computer and the Ultra96—over ethernet. Your computer will send packets at a fixed rate. The Ultra96 will receive the data and compress it. Figure 5.5 from homework 5 shows you the setup and cabling. Since the first system is sending data at a fixed rate, it is necessary for the receiver to compress the data at that rate or data will be lost. We provide the code for the sender (`Client/client.cpp`). Your project is connected to the receiver (`Server/encoder.cpp`). And then you can use the decoder (`Decoder/Decoder.cpp`) to verify that you can recover the original, unencoded file from the compressed file.

6. Let's run the given code with the system we have setup. After compiling the code, copy over `encoder` binary, and the `LittlePrince.txt` as follows (adjust the commands if you are not using Linux):

   ```
   scp encoder LittlePrince.txt root@10.10.7.1:~/
   ```

   And then open the Ultra96 terminal and run the encoder with `./encoder`. The program waits for a packet to arrive. Open a terminal in your computer and issue the following command:

   ```
   ./client -i 10.10.7.1 -f LittlePrince.txt
   ```

   You should see the following output in your Ultra96 terminal:

```
root@ultra96v2-2021-1:~# ./encoder
setting up sever...
server setup complete!
write file with 14247
--------------- Key execution times ---------------
Reading packets and processing :    0.228 ms
```

You should see the following output in your host terminal:

```
ip is set to 10.10.7.1
filename is LittlePrince.txt
bytes_read 14247
```

You can verify the output by doing the following in the Ultra96:

```
diff output_cpu.bin LittlePrince.txt
```

Note that our example is just writing the packets to a file. Your project will process these packets with the encoder pipeline and write a compressed output.

7. We'll now describe what's happening in the client and the server:

   (a) **Packet Layout**: We will be sending data via UDP datagrams. Linux supports the UDP protocol and receiving packets from the client can be done easily using Linux IP. The code provided will direct you on how to setup your compression pipeline to listen as well as handle incoming packets. The maximum size of a packet will be 16K Bytes. The header of the packet will be 2 bytes consisting of a done bit denoting that all of the data has been transmitted as well as the length of the data contained inside the packet.

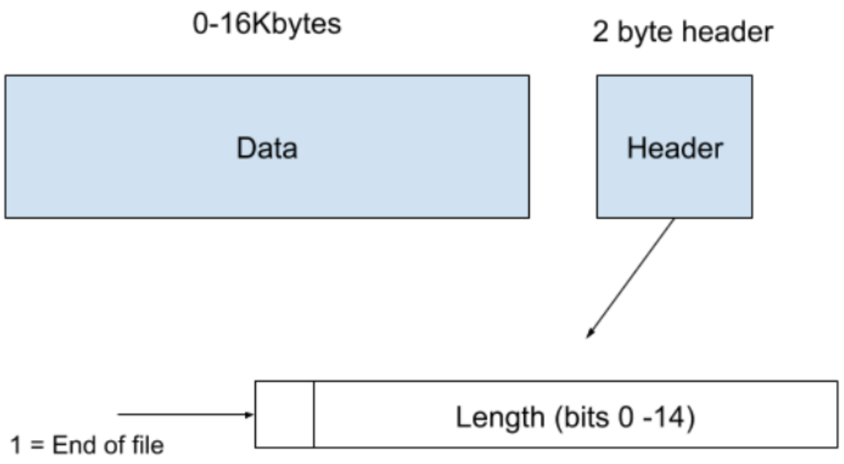


Figure 1: Packet Layout

(b) You will be specifically interested in the files `Server/server.cpp` and `Server/server.h`. These are the files provided and can be directly copied and pasted into your project to set up your server pipeline to receive data. The rest of the repository can serve as an example of how one could implement the design (`Server/encoder.cpp`). At a high level you need to call the function `setup_server()` once. Following this you can makes calls to the function `getpacket()` to receive your next datagram. Please note that your buffer passed into this function needs to be able to handle the maximum payload size plus the two byte header. Please also note that the `recvfrom()` Linux call is blocking. This means that your process will be blocked until a datagram has arrived. Depending on your design this may be ok. If you do not want to block you may look into the `select()` system call as an alternative. This will allow you to check if data has arrived in your socket and take actions accordingly.

You are of course free to write your own application. For more information on how to receive the data you can refer to the man page. https://linux.die.net/man/2/recvfrom.

Alternatively, you can create your own client and server using the DPDK library. Examples of how to write client and server code using DPDK can be found in the following links:

- https://doc.dpdk.org/guides/index.html
- https://zenhox.github.io/2018/01/25/dpdk-pktSR/

15.3 Design Considerations

1. **Configuring Sender**: You will likely need to slow down the client's data transfer to begin with. If your system cannot keep up with the pace of the sender, packets will be dropped.

   To configure the sender, when you start the client from the Linux shell, it takes arguments as shown:

   ```
   ./client -s 5 -f file -i ip_address_of_ultra96 -b blocksize
   ```

   Usage example is:

   ```
   ./client -s 5 -f LittlePrince.txt -i 10.10.7.1 -b 2048
   ```

   `-s` option specifies the sleep time or delay between packets (in microseconds)
   `-i` option specifies ip address to send to
   `-f` option specifies what file to send
   `-b` option specifies the block size

   For the project report and for project milestones where you characterize your throughput, you should adjust the `-s` argument until your design fails. Report your maximum throughput as the throughput associated with the smallest value of `-s` on which your design successfully receives and correctly compresses the input. Measure the actual throughput by measuring the time it takes for the client to send the file. You can use `/usr/bin/time` to measure the time.

2. **Debugging Sending and Receiving of Packets**: If you encounter problems with sending and receiving packets between the client and the server, you can emulate the socket programming. You can see an example of that here. Specifically in `server.cpp` and `server.h`, you can see that instead of using sockets, you can read a file and send it in pieces to the encoder pipeline to emulate socket programming.