Precise and Closed-loop Traffic Generation with Caliper "Technical Documentation"

Monia Ghobadi, Goeff Salmon, Yashar Ganjali, Martin Labrecque, J. Gregory Steffan

May 2010

This document describes all the parts of Caliper: a Precise and Closed-loop Traffic Generator in detail. The individual pieces are described in detail. Useful background and related material is presented in the appendix.

Contents

1 Overview

Caliper's main objective is to precisely control the transmission times of packets which are created in the host computer, continually streamed to the NetFPGA, and transmitted on the wire. The generated packets are sent out of a single Ethernet port of the NetFPGA, according to any given sequence of requested inter- transmission times. It is important to note that because packets are streamed, the generator can immediately change the traffic in response to feedback. Unlike previous works that replay packets with prerecorded transmission times from a trace file, Caliper generates live packets and supports closed-loop traffic. Therefore, Caliper can easily be coupled with existing traffic generators (such as Iperf and Netperf) to improve their accuracy at small time scales.

Caliper is built on NetThreads, a platform we have created for developing packet processing applications on FPGA-based devices and the NetFPGA in particular. NetThreads is primarily composed of FPGA-based multithreaded processors, providing a familiar yet exible environment for software developers: programs are written in `C`, and existing applications can be ported to the platform. In contrast with a PC or NIC-based solution, NetThreads is similar to a custom hardware solution since it offers direct network I/O and allows the programmer to specify accurate timing requirements.

Caliper has 3 main parts: the userspace packet generator, the kernel module, and the NetThreads application. Here's the life-cycle of an outgoing packet. First the userspace packet generator decides a packet should be sent at a particular time. A description of the packet, containing the transmission time and all the information necessary to assemble the packet is sent to the kernel module, which is also the NetFPGA driver. In the driver, multiple packet descriptions are combined together and copied to the NetFPGA. Combining descriptions reduces the number of separate transfers required and is necessary for sending packets at the line rate of 1Gb/s. From there, the packet descriptions are each given to a single thread running on the NetThreads soft-processors. Each thread will assemble its packet in the NetThreads' output memory. Next, a single thread sends all of the prepared packets in the correct order at requested transmission time. Finally, the hardware pipeline of the NetFPGA sends the packets onto the wire.

2 Userspace Packet Generator

This is a normal userspace process that tells Caliper what packets to send and when to send them. It communicates with the NetFPGA driver using NetLink (See section 6.1). It is written in C++, and the current code is in `http://www.cs.toronto.edu/~monia/precisegen_1.1.tar.gz` in the directory `pktpasser/pktgen`. The Makefile in the directory produces the `pktgen` executable, which has the following command-line options:

-n NUM Controls how many packets will be sent in total.

-t TIME Controls how long long packets will be sent for.

-d DELAY Sets the delay between sending each packet. Units is nanoseconds.

-D FILE Sets a file to read delays from. FILE is a text file with delay amounts in nanoseconds separated by white space. The delays are used sequentially, and once the last delay is used pktgen will continue from beginning of the file.

-l PAYLOAD Sets the size of the UDP payloads in the packets sent. Units is bytes.

-L FILE Sets a file to read payload sizes from. FILE is a text file with UDP payload sizes in bytes separated by white space. The sizes are used sequentially, and once the last delay is used pktgen will continue from beginning of the file.

-i DST_IP Chooses the destination IP to send packets to.

-I SRC_IP Chooses the source IP to set in packets.

-m DST_MAC Sets the destination MAC address to send packets to.

-N Decides whether packets will be sent through the NetFPGA. If this option is not present then pktgen will send packets through the normal network stack and use busy-waiting to time the delays between packets. Some options, such as -m and -I do not work unless this option is present.

-h Prints help message.

Some of the above options conflict with each other. Both -n and -t cannot be present. The inter-transmission delay must be set with either -d or -D, but not both. Similarly, -l and -L cannot both be used to set the payload length.

3 NetFPGA Driver

The driver used by the packet generator is a modified version of the original NetFPGA driver. The current code is in `http://www.cs.toronto.edu/~monia/precisegen_1.1.tar.gz` in the directory `pktpasser/driver/kernel`. The Makefile in that directory will build the kernel module nf2.ko. It also contains the target `modprobe` which will build, install and replace the currently inserted kernel module with the new one. For comparison, it's useful to also understand the original NetFPGA driver.

Like every other Linux module, the driver is written in C. Section 6.2 contains general information about programming in the Linux kernel. The rest of this section describes the modified driver specifically.

The main task of Caliper's NetFPGA driver is to receive packet descriptions over NetLink (see section 6.1), assemble real packet headers, combine them together and copy them to the NetFPGA card using DMA over the PCI bus. It avoids overflowing buffers within the NetFPGA by waiting for explicit feedback before sending the packet headers.

Getting familiar with the driver code takes some time. Here are the three most important source files (all are in the `pktpasser/driver/kernel` directory):

nf2kernel.h Defines important structs that hold the driver's state.

nf2main.c Has the modules entrance and exit functions, sets up hooks for the PCI bus, and handles receiving NetLink messages.

nf2_control.c Initializes and communicates with the NetFPGA card. This is where most of the real work is done.

All communication between the driver and the NetThreads packet generator goes over the PCI bus. Both the driver and the NetFPGA can DMA transfer messages to each other. The types of messages are defined in drivercomm.h. Note there is an identical drivercomm.h file in the source code of the NetThreads packet generator (see Section 5). If one is changed, the other must be updated. Unfortunately, they are in two separate SVN repos so this must be done manually for now.

Here are the main concepts and execution paths present in the driver.

3.1 Transmit Buffers: txbuff

At initialization, the driver allocates a fixed number of txbuff structs. Each one stores multiple packet headers that are waiting to be sent to the NetFPGA. As soon as a DMA transfer of a txbuff to the NetFPGA has completed, the txbuff is added back to the free list.

Headers for multiple packets are combined together because each DMA transfers incurs some overhead. It is not possible to send each packet header individually fast enough. For example, when using the NetFPGA as a normal NIC, the total throughput is only 260Mb/s when individually transferring 1518 byte packets.

Each of the allocated txbuffs is always in one of three states: (i) it is unused and is in the `free_txbuffs` list, (ii) it contains some packet headers but still has room for more then it is pointed to by the `txbuff_building` pointer, or (iii) it is waiting to be copy to the NetFPGA card and is in the `transmit_txbuffs` list. The txbuff at the head of the `transmit_txbuffs` list is either currently being copied or will be copied at the first opportunity.

3.2 Receiving Packets to Send Via NetLink

Each arriving NetLink message is sent to the function `nl_receive_msg` in nf2main.c. If it contains packet descriptions it is passed to `nf2c_inject_pkt` in nf2_control.c, which builds the actual packet header and calls `nf2c_tx_pkt`. This function adds the packet header to a txbuff struct. If there is a partially filled txbuff with room for the packet header, it will be used. Otherwise, the function gets a new txbuff from the `free_txbuffs` list. Next, `nf2c_attempt_send` is called which will start a DMA transfer to the NetFPGA if none is currently in progress. At most one DMA transfer can be occurring at any one time.

3.3 Handling Interrupts

To communicate with the driver the NetFPGA card can raise an interrupt, which causes `nf2c_intr` to be called. This function reads a status register of the NetFPGA to determine why the interrupt was raised and handles it appropriately. Here are the most interesting interrupts:

INT_DMA_TX_COMPLETE Tells the driver that a DMA to the NetFPGA has completed. The driver will call `nf2c_attempt_send` to start sending the next waiting txbuff, if appropriate

INT_PKT_AVAIL Tells the driver that the NetFPGA has a packet ready to DMA to the driver.

INT_DMA_RX_COMPLETE Occurs when the NetFPGA has finished sending a packet to the driver. This then calls `nf2c_rx` with the actual packet.

3.4 Transmit Quota

NetThreads and the NetFPGA have a limited amount of buffering available. If the driver sends too many packets to the NetFPGA, then they will be dropped at the input queues in the NetFPGA. To avoid this, the driver maintains a quota of the number of packets it can send to the NetFPGA. The `allowed_send` variable stores the number of packets that the driver can send. If it reaches zero, then the driver will stop doing DMA transfers. When the driver receives a quota message from the NetFPGA, it increases `allowed_send`. The logic for this is in the function `nf2c_rx`.

3.5 Marking Packets

Marking is a feature added to avoid using all of the available txbuffs in the driver and dropping packets. It provides explicit acknowledgement back to a process that is sending packets to the driver.

A process can send a NetLink message to the driver to set a mark. When the driver receives the message, the function `nf2c_mark` is called which sets some fields of the txbuff containing the last packet that the driver received over NetLink. When that txbuff is eventually copied to the NetFPGA, the driver will send a NetLink message to the process that originally set the mark (see the `notify_of_mark` function).

3.6 Starting and Resetting

When nf2 module is first inserted, either when Linux boots or when you run modprobe explicitly, the network interface it creates will be down and the driver will not send packets to the NetFPGA and will ignore interrupts.

Bring the interface up by running `ifconfig nf2c0 up`, which causes the driver function `nf2c_open` to be called. The function initializes the NetFPGA card and sends a reset command message to the NetFPGA. In response, the

NetThreads packet generator will respond with another reset message. Before this reply is received from the NetFPGA, no packets will be sent to or from the NetFPGA.

Bring down the interface with `ifconfig nf2c0 down`. This clears all packets that are waiting in the driver to be sent and ignores any future interrupts from the NetFPGA.

3.7 Multiple Network Interfaces and NetFPGA Cards

The original NetFPGA driver creates four network interfaces in Linux and supports multiple NetFPGA cards in a single computer. For the first card, it would create interfaces nf2c0, nf2c1, nf2c2, and nf2c3. For the second card, it would create nf2c4, nf2c5, and so on. The modified driver for packet generation creates only a single network interface. Also the modified driver only supports a single NetFPGA card in the machine. This is partly because it simplified the API for sending packets (you don't need to pick an interface when sending a packet, there is only one), and partly because I did not test with multiple NetFPGA cards in a single computer.

Sending packets received over NetLink requires the driver get access to the state for the NetFPGA. Currently this is simply stored in the global variable `inject_dev`. To support multiple NetFPGA cards, the driver must select and use the state for the correct NetFPGA instead.

4 NetThreads

NetThreads is a platform for packet processing on the NetFPGA. It lets you write C programs that run on the NetFPGA. The programs are cross-compiled and executed by soft-processors instantiated in the FPGA. By using NetThreads you can create applications for the NetFPGA without having to write a line of Verilog.

4.1 Architecture

NetThreads has two CPUs, each of which has 4 threads. There is only a small software library and no operating system. Unlike threads and processes in a normal CPU, there is no context-switch overhead between the threads. They execute in round-robin order, with each thread executing one instruction every 4 clock cycles. The CPUs use network byte-order regardless of the endian-ness of the host CPU.

The CPUs in NetThreads fit into the same hardware pipeline used by other NetFPGA applications like the NIC and router. A description of the pipeline here `http://netfpga.org/netfpgawiki/index.php/Guide#Reference_Pipeline` also applies to NetThreads. In the diagram of the pipeline (`http://netfpga.org/netfpgawiki/index.php/Image:ReferencePipeline.jpg`), the NetThreads CPUs replace the Output Port Lookup module and sit between the Input Arbiter and the Output Queues. In the reference pipeline, there are 4 input queues (labelled "CPU RxQ" in the diagram) for packets copied over the PCI bus from the host computer's CPU. However, in NetThreads, only one of these queues is connected to the input arbiter. The remaining 3 are never used and packets sent to them by the driver will never be read by NetThreads.

4.2 Software

Here is an overview of the important parts of the source code:

netfpga The root of the svn repo

- **/threads** The simulator: good for testing NetThreads applications without a NetFPGA
- **/bench** Contains software that runs in NetFPGA
  - **/common** library used by all NetThreads applications

/pktgen packet generator application
/bounce simple application that resends all received packets

NetThreads applications have multiple threads with a single entry point. Threads are not dynamically created or destroyed. Instead, they all begin executing the same function at the same time. To change the behaviour of a specific thread, call `nf_tid` to get the current thread's unique thread id and branch based on the result.

Listing 1: netfpga/simple/main.c

```
#include "support.h"

int main(void) {
  if (nf_tid() == 2) {
    log("Hello world. I am the wonderful thread 2\n");
  } else {
    log("Hello world. I am thread %d\n", nf_tid());
  }
  return 0;
}
```

4.3 Build System

Behind the scenes, the build system for NetThreads applications is a bit messy. Fortunately, the complexities are mostly hidden and the individual Makefiles in the application directories are relatively simple. Here is bounce's Makefile:

Listing 2: netfpga/bounce/Makefile

```
TARGETS=bounce
include ../bench.mk
bounce: process.o pktbuff.o memcpy.o
```

The TARGETS variable should contain the names of the one or more resulting binaries, and the variable must be set before including bench.mk. Currently the Makefiles must be in a directory directly beneath the bench directory because of problems with the paths to tools (I think the MAKEFILE_LIST variable could be used to fix this, if needed). For each target in

TARGETS, the makefile must explicitly say what object files the application depends on. These objects files, plus one or two other default ones, are linked together to build the binary. Note that although bounce depends on process.o, pktbuff.o, and memcpy.o, only process.c is in the bounce directory. The other source files, pktbuff.c and memcpy.c, are in bench/common. Any object file that builds from a source file in the common directory can be used in this way. Also, the headers in the common directory can be included in source files as if they were local. It's not necessary to add "../common/" to the paths.

NetThreads applications can be compiled for three different contexts:

nf Builds the app to run on the NetFPGA. This is the default context. All calls to the `log` function are ignored.

sim Builds the app to run in the simulator. Actually, the resulting binary should also run on the NetFPGA. The main difference is that the `log` function is supported and will print to standard out when using the simulator.

sw Builds the app to run as software on the host computer. Instead of compiling using a cross-compiler this uses the native gcc toolchain. A lot of the normal NetThreads functions like sending or receiving packets are noops in this context. This is probably only useful when porting existing applications to NetThreads.

Select the context by passing the name of the context to make, ex. `make CONTEXT=sim`. The default context is `nf`. When switching between contexts, run `make clean` to ensure all object files are recompiled.

4.4 NetThreads API

The bench/common directory contains functions and structs useful for writing NetThreads applications. This document describes the most important and commonly used functions, but the best and most up-to-date source of information is still the code itself.

4.4.1 Misc Functions

uint nf_time()

Returns the current NetFPGA time. It increments once every clock cycle at a frequency of 125MHz. The time is returned as a 32-bit number, so it will wrap around to zero roughly every 34 seconds.

int nf_tid()

Returns the current threads unique id. The id is a number in the range [0, numthreads - 1].

void log(char *frmt, ...)

Prints a string. The arguments of log are the same as printf's. This function only has an effect in the sim and sw contexts. In the nf context, the function is defined away.

4.4.2 Receiving Packets

NetThreads places arriving packets into the input memory.

void nf_pktin_init()

Initializes NetThreads for receiving packets by dividing the input memory into tens slots of 1600 bytes each. Must be called at most once from a single thread.

t_addr* nf_pktin_pop()

Checks if a packet has been received. Arriving packets will be returned by nf_pktin_pop only once and are returned in the order they arrived. If a packet is waiting, then this function returns a pointer to the IOQ header at the start of the packet. Use nf_pktin_is_valid to determine if a packet is actually returned.

int nf_pktin_is_valid(t_addr* addr)

Determines if a pointer returned by nf_pktin_pop is actually a packet or not. Returns true if the pointer is a valid packet, false otherwise.

void nf_pktin_free(t_addr* val)

Tells NetThreads that the application has finished reading a packet returned by nf_pktin_pop. After calling this function, an application should not read the packet contents again. It is important to call this function as soon as possible. If packets in the input memory are not freed then arriving packets cannot be stored, which quickly leads to packet drops.

4.4.3 Sending Packets

To send packets an application must fill in the packet headers and payload in the NetThreads output memory. The hardware can only send packets that are stored in the output memory. To send a received packet, the application must copy the packet from the input memory to the output memory.

void nf_pktout_init()
: Initializes NetThreads for sending packets. Must be called only once from a single thread.

t_addr* nf_pktout_alloc(uint size)
: Allocates space in the output memory for a packet. Currently the argument `size` is ignored and all spaces returned are 1600 bytes long. The returned address is either a pointer to the newly allocated space or 0 if no space is available.

void nf_pktout_send(char* start_addr, char* end_addr)
: Sends a packet. The argument start_addr points to the start of the packet and end_addr points to the byte after the last byte of the packet.

4.4.4 IOQ Headers

All packets received and sent by NetThreads start with an 8 byte header that is added and removed by the NetFPGA itself and does not exist on the wire. This header is called the IOQ Modules Header (or just IOQ Header) and is described here `http://netfpga.org/netfpgawiki/index.php/Guide#Reference_Pipeline_Details`.

The IOQ header specifies both the length of a packet and which NetFPGA port it was received on and which port it will be sent to. The file bench/-common/pktbuff.h contains the following for working with IOQ headers:

struct ioq_header
: The actual in-memory layout of the header.

void fill_ioq(struct ioq_header *ioq, unsigned short port, unsigned short bytes)
: Fills the IOQ Header of a packet that will be sent. It does not set the source port of the packet because the hardware for sending packets ignores the source port field.

4.4.5 Locks

NetThreads offers 16 mutexes for synchronizing between threads. Each mutex or lock is identified by a integer between 0 and 15 (higher numbers simply wrap around and identify the same 16 locks).

void nf_lock(int id)
  Acquires a lock.
void nf_unlock(int id)
  Releases a lock.

Think of the locks as 16 booleans. If a lock is false, then nf_lock will set the lock true and return immediately. If a flag is already true, then nf_lock will block the calling thread until the lock is false. Calling nf_unlock sets a lock false and never blocks.

4.5 Simulator

The NetThreads simulator runs NetThreads applications without using a NetFPGA. It simulates the CPUs and hardware of NetThreads and can send and receive packets to and from a number of sources.

The simulator is in the netfpga/threads directory and is compiled by running `make`. The number of CPUs and threads is set in config.h.

To simulate an app, pass the binary to the simulator, ex. `./trace ../bench/simple/simple`. The simulator has lots of options (run `./trace -h` and `./trace -help` to see some).

Testing applications usually requires the application can receive and send packets. There are some useful options for getting packets into the simulator. The -macN and -hostN options connect the ports in the simulated NetFPGA to tap network interfaces, which allows real packets to be sent to or from the simulator. The -pcapmacN, and -pcaphostN read packets from a pcap file and send them to input ports of the simulated NetFPGA. Also, see the init_test_pkts() in tracing.cc to see examples of programatically injecting packets.

4.6 Loader

After compiling an application in the nf context, run `make mif` to create *.instr.mif and *.data.mif, which contain the application's instruction and data segments, respectively. These files can then be loaded onto NetThreads. The loader program in netfpga/loader will take this files, download the application to the NetFPGA and start the application running.

Steps to run a NetThreads application:

1. Reprogram the CPCI. This needs to be done once every time the computer is booted (doing it more often doesn't hurt though). Do not use the latest cpci_reprogrammer.pl file that is in the server's PATH. The NetThreads system was built against the Verilog files from an older version of the NetFPGA distribution, and it requires a matching CPCI bitfile. Do this by running `cpci_reprogrammer.pl`

2. Download the NetThreads bitfile to the NetFPGA.
   Run `nf2_download netthreads1.bit`.

3. Download the application's instructions.
   Run `pathtopktgen/loader -i app.instr.mif`
   replacing "app" with the correct name.

4. Download the application's data.
   Run `pathtopktgen/loader -d app.data.mif -nodebug`
   replacing "app" with the correct name.

As soon as the data is loaded the application will begin running.

5 NetThreads Packet Generator

This section describes the Packet Generator NetThreads application, called the pktgen hereafter. The pktgen receives packets sent from the NetFPGA driver containing multiple packet headers and builds and sends packets out of one Ethernet port at the appropriate times. The pktgen code is in `pathtonetthreads/pktgen/trunk/netfpga` in the bench/pktgen directory. After installing the NetThreads tools as described in Section 4.1, run `make` to compile the pktgen.

5.1 Thread Roles

There are eight threads in NetThreads. To easily send packets in the correct order and at the correct time, only one thread ever sends packets in the pktgen. This thread has thread id 0 and is called the *sending thread*. Near the end of the `main` function the sending thread calls `manage_sending`, which never returns, while the other seven threads serve jobs from a work queue.

```
if (nf_tid() == 0) {
  manage_sending();
} else {
  workq_serve(&jobs_queue, -1);
}
```

There are only two types of jobs in the work queue:

1. Receive a packet by calling the `pop_work` function;
2. Prepare a packet in the output memory by calling the `prepare_work` function.

Each packet that the NetFPGA driver sends to the NetFPGA contains the descriptions of multiple packets to send. Eventually a NetThreads thread will pop the packet and call `process_pkt_pkts`, which examines the packet descriptions and adds a job for each description to the `jobs_queue`. Next threads will pop the jobs from the queue and begin copying the packet headers from the descriptions in the input memory to free slots in the output memory. Once the packets to send are prepared, they are given to the sending thread in the order they should be sent. The time it takes to prepare each packet may vary so only the thread that has finished preparing the packet that is scheduled to be sent first will notify the sending thread. See the variable `next_pkt_tosend` and the circular array `prepared_pkts` for how this is done.

The sending thread spends all its time looping in the `manage_sending` function. It continually checks for newly prepared packets to send. Once it has a packet ready, it continually calculates how much time is left before the packet's scheduled departure. When the departure is imminent (currently that means it's within 5000 clock-cycles) it calls `tight_send_loop` to actually send the packet. As its name suggests, originally this function would continually loop checking the time. However, it now uses a hardware-triggered send by calling `nf_pktout_send_schedule`. This tells the hardware when to send the packet and is more accurate than the previous method. If the time has already passed then the packet will be sent immediately. Otherwise, the hardware waits until the clock reaches the schedules time before passing the packet to the rest of the NetFPGA pipeline.

5.2 Work Queues and Multi-threaded Deques

The pktgen uses two handy data structures: the `mtdeque` and the `work_queue`. Both are in the the bench/common directory. In brief, the mtdeque is a singly linked-list that multiple threads can push or pop from at the same time. The work_queue uses an mtdeque to store function pointers and data which represent a piece of work for a thread to do. Different threads serve the queue by popping off work and passing the data to the specified function. See the header files `mtdeque.h` and `workqueue.h` for documentation of each relevant function.

5.3 Managing Input and Output Memory

The input and output memories in NetThreads are 16KB each. In the pktgen, as in most NetThreads applications, both memories are divided into ten 1600 byte slots. The pktgen must carefully manage slots in both.

As described in 3.4, the NetFPGA driver avoids filling all of the input memory slots by waiting for explicit feedback from the pktgen. The feedback is a quota packet sent by the sending thread from the function `send_special`.

Once an packet is popped the input memory slot cannot be freed until the packet is no longer needed. Multiple threads can be reading different packet descriptions from the same packet at the same time. To avoid freeing the packet too early, or multiple times, there is a reference count in the struct associated with each input memory slot. See the end of the function `prepare_work` where it decrements the count and optionally calls `free_input_pkt`.

Each slot in the output memory used to send packets is represented by a `pkt_out` struct which holds information about the slot and is used to schedule the `prepare_work` jobs in the work queue. Empty slots are stored in the `free_pkts` queue. Only 8 slots are used to send packets to the Ethernet ports. The remaining 2 are used to send feedback quota packets back the the NetFPGA driver. The following code in the `main` function allocates the output memory slots for these two purposes:

```
quota_pkt = nf_pktout_alloc(1520);

/* Add all pkt_out structs to the free list */
mtdeque_lock(&free_pkts);
for (i = 0; i < NUM_PKT_OUTS; i++) {
  struct pkt_out *po = pkt_outs + i;
  po->output_mem = nf_pktout_alloc(1520);
  mtdeque_push_nolock(&free_pkts, &po->item);
}
mtdeque_unlock(&free_pkts);
```

One slot is assigned to the `quota_pkt` and 8 are added to `pkt_out` structs and added to the `free_pkts` queue. The number 1520 passed to `nf_pktout_alloc` is meaningless and ignored. Note that this is the only time that the `nf_pktout_alloc` function is called in pktgen. Unlike most NetThreads applications, pktgen explicitly collects the previously sent packet when sending a new packet. Compare the `nf_pktout_send_finish_nocollect()` in pktgen.c with `nf_pktout_send_finish()` in support.c. Collecting the free packet directly is more efficient that going through the normal `nf_pktout_alloc` mechanism, and if the packets are scheduled to depart close together then the sending thread cannot afford to waste clock cycles.

5.4 Time

The sending thread maintains the current time in two global variables: `time_hi` and `time_lo`. It stores the last value returned by `nf_time()` in `time_lo` and counts the number of times that the 32-bit time value wraps in `time_hi`. Together these variables store a 64-bit time value with a granularity of 8ns which moves ahead to the current time whenever `update_time()` is called. The `time_until` function returns the time between the current time and a given time.

Their are two ways to specify a packet's departure time. Either the time can be the absolute time or a relative time from the last packet transmission. Currently, the userspace packet generator in Section 2 sends the first packet with an absolute time and all subsequent packets with relative times delays, but there is nothing in the driver or pktgen that requires this.

6 Appendix

6.1 NetLink

Linux's NetLink allows userspace and kernel code to communicate using a socket-like API. See this article `http://www.linuxjournal.com/article/7356` for a nice intro to NetLink. Note that it describes an older kernel than what we use, so the structures and functions have changed slightly. For more up to date info see `man 3 netlink` and `man 7 netlink`. Also look at the code in both the userspace packet generator (Section 2) and the driver (Section 3) for examples of using NetLink.

6.2 Linux Kernel Programming

Modifying code in the Linux kernel can be difficult at first. The environment is different from normal userspace code. The functions and APIs that are available are different. Code runs in one of several contexts (process or interrupt context) where the rules for what you can and can't do change. Finally, bugs and mistakes can lock up the machine and require a reboot.

A great resource, especially for modifying NetFPGA's driver is the book Linux Device Drivers, 3rd Edition, available online either in PDF (`http://lwn.net/Kernel/LDD3/`) or HTML (`http://www.makelinux.net/ldd3/`). Chapters 2, 5, 7, and 17 are very helpful. The others contain useful info too.

The LXR website `http://lxr.linux.no/linux` is useful for navigating the kernel source. It shows you where functions and symbols are defined and where they are used in specific kernel versions. It's great for seeing how a particular function is used in other parts of the kernel.

The best way to learn more is probably reading the driver code and comparing the original NetFPGA driver with the modified Caliper NetFPGA driver.

A friendly warning: watch out for printk. It's the kernel's equivalent to printf and is handy for debugging and error messages. However, it's very easy to call printk too often and significantly slow down the computer which leads to performance problems for the packet generator. This is especially true if the printk calls are executed in interrupt context.