Analysis of NIC and Test of PCI Multi-Bus Simulator

Student Name: Ligang Wang

Advisor Name: Dr. Mike MacGregor

September 2007

Acknowledgments

I would like to thank my supervisor, Dr. Mike MacGregor, for his continuous support and help with patience. Special thanks for a lot of helpful discussions to Kevin Ye and our team, Liao and Sun. Last but not least, I would like to thank my wife, Rong and my new born baby, Kelly for all their support and love during this time.

Ligang Wang
September 2007

Abstract

This thesis presents the analysis of 10 Gigabit Ethernet Network Interface Card and simulating & testing of virtual NIC, which is defined as the device in our PCI Multi-Bus Simulator. Network interface processing requires support for the following characteristics: a large volume of frame data, frequently accessed frame metadata, and high frame rate processing. So we also discuss how to improve network interface efficiency. By using Gnuplot tools, we can plot some diagrams which are helpful to my data analysis. Since the virtual NIC in our PCI Multi-Bus Simulator is just a simple simulation, a NIC Memory Control model's algorithm was developed.

Table of Contents

1. Specification of PCI Bus……………………………………………………5

1.1 Introduction to PCI Bus……………………………………………………5

1.2 Architecture of PCI Bus……………………………………………………5

2. Analysis of NIC…………………………………………………………...7

2.1 Introduction to NIC…………………………………………………………7

2.2 Architecture of NIC…………………………………………………………7

2.2.1 Basic NIC Architecture…………………………………………………...7

2.2.2 10 Gigabit NIC Architecture……………………………………………...8

2.2.3 10 Gigabit NIC Memory System…………………………………………9

2.2.3.1 Multiple processing cores…………………………………………...10

2.2.3.2 Data Memory………………………………………………………...11

2.2.4 Send and receive of NIC …………………………………………………12

2.2.5 Hardware Parallelism of NIC……………………………………………14

2.2.5.1 Previous Hardware Parallelism ……………………………………...15

2.2.5.2 Modern Hardware Parallelism………………………………………16

2.3 Simulating and testing data of virtual NIC…………………………………17

2.3.1 Testing data and plotting…………………………………………………17

2.3.1.1 Single bus simulator…………………………………………………17

2.3.1.2 Two bus simulator……………………………………………………21

2.3.1.3 Data collection of Multi-Bus Simulator……………………………...24

2.3.2 NIC Memory Control (MC) model's algorithm…………………………..27

References……………………………………………………………………...29

Appendix………………………………………………………....................30

List of Figures and Plots

Figure1. PCI System Architecture……………………………………………………5

Figure2. Basic NIC Architecture……………………………………………………..8

Figure3. 10 Gigabit NIC Architecture………………………………………………..8

Figure4. Steps involved in sending a packet…………………………………………12

Figure5. Steps involved in receiving a packet……………………………………….13

Figure6. Previous hardware parallelism (with an event register)……………………15

Figure7. Modern hardware parallelism (with a distributed event queue)……………16

Plot1.Data rate (single bus)…………………………………………………………..18

Plot2. Throughput plot (single bus)………………………………………………….19

Plot3. Transfer time histogram (single bus)…………………………………………20

Plot4. Data rate (two buses)…………………………………………………………22

Plot5. Throughput plot (two buses)………………………………………………….23

Plot6. Transfer time histogram (two buses)………………………………………….24

1. Specification of PCI Bus

1.1 Introduction to PCI Bus

The PCI (Peripheral Component Interconnect) local bus is a high speed bus. The PCI Bus was proposed at an Intel Technical Forum in December 1991, and the first version of the specification was released in June 1992. The current specification of the PCI bus is revision 3.0, which was released on February3, 2004. Since its introduction, the PCI bus has gained wide support from all the computer industry. Almost all PC systems today contain PCI slots, as well as the Apple and IBM Power-PC based machines, and the Digital Alpha based machines. The PCI standard has become very popular. The PCI Bus is designed to overcome many of the limitations in previous buses. The major benefits of using PCI are: (1) High speed (2) Expandability (3) Low Power (4) Automatic Configuration (5) Future expansion (6) Portability (7) Complex memory hierarchy support (8) Interoperability with existing standards.

PCI is also called an intermediate local bus, to distinguish it from the CPU bus. The concept of the local bus solves the downward compatibility problem in an elegant way. The system may incorporate an ISA, EISA, or Micro Channel bus, and adapters compatible with these buses. On the other hand, high-performance adapters, such as graphics or network cards, may plug directly into PCI. PCI also provides a standard and stable interface for peripheral chips. By interfacing to the PCI, rather than to the CPU bus, peripheral chips remain useful as new microprocessors are introduced. The PCI bus itself is linked to the CPU bus through a PCI to Host Bridge.

1.2 Architecture of PCI Bus

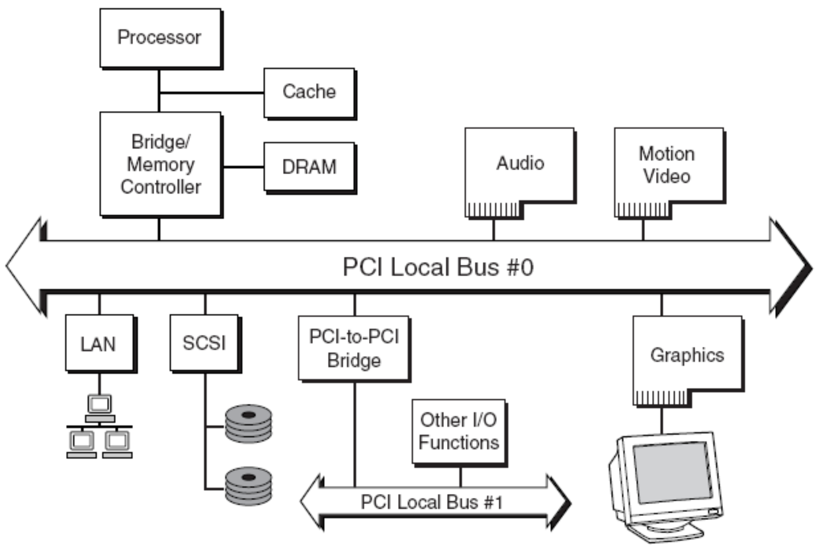


Figure1. PCI System Architecture

Figure1 shows a typical PCI Local Bus system architecture. This example is not intended to imply any specific architectural limits. In this example, the processor/cache/memory subsystem is connected to PCI through a PCI bridge. This bridge provides a low latency path through which the processor may directly access PCI devices mapped anywhere in the memory or I/O address spaces. It also provides a high bandwidth path allowing PCI masters direct access to main memory. The bridge may include optional functions such as arbitration and hot plugging. The amount of data buffering a bridge includes is implementation specific.

The basic PCI transfer is a burst. This means that all memory space and I/O space accesses occur in burst mode; a single transfer is considered a “burst” terminated after a single data phase. Addresses and data use the same 32-bit, multiplexed, address/data bus. The first clock is used to transfer the address and bus command code. The next clock begins one or more data transfers, during which either the master, or the target, may insert wait cycles.

PCI supports posting. A posted transaction completes on the originating bus before it completes on the target bus. For example, the CPU may write data at high speed into a buffer in a CPU-to-PCI bridge. In this case the CPU bus is the originating bus and PCI is the target bus. The bridge transfers data to the target (PCI bus) as long as the buffer is not empty, and asserts a not ready signal when the buffer becomes empty. In the other direction, a device may post data on the PCI bus, to be buffered in the bridge, and transferred from there to the CPU via the CPU bus. If the buffer becomes temporarily full, the bridge de-asserts the target ready signal.

In a read transaction, a turnaround cycle is required to avoid contention when the master stops driving the address and the target to begin driving the data on the multiplexed address/data bus. This is not necessary in a write transaction, when the master drives both the address and data lines. A turnaround cycle is required, however, for all signals that may be driven by more than one PCI unit. Also, an idle clock cycle is normally required between two transactions, but there are two kinds of back-to-back transactions in which this idle cycle may be eliminated. In both cases the first transaction must be a write, so that no turnaround cycle is needed, the master drives the data at the end of the first transaction, and the address at the beginning of the second transaction. The first kind of back-to-back occurs when the second transaction has the same target as the first one. Every PCI target device must support this kind of back-to-back transaction. The second kind of back-to-back occurs when the target of the second transaction is different than the target of the first one, and the second target has the Fast Back-to-Back Capable bit in the status register set to one, indicating that it supports this kind of back-to-back.

For arbitration, PCI provides a pair of request and grant signals for each PCI unit, and defines a central arbiter whose task is to receive and grant requests, but leaves to the designer the choice of a specific arbitration algorithm. PCI also supports *bus parking*, allowing a master to remain bus owner as long as no other device requests the bus. The default master becomes bus owner when the bus is idle. The arbiter can select any master to be the default owner.

PCI provides a set of configuration registers collectively referred to as “configuration space.” By using configuration registers, software may install and configure devices without manual switches and without user intervention. Unlike the ISA architecture, devices are re-locatable - not constrained to a specific PCI slot. Regardless of the PCI slot in which the device is located, software may bind a device to the interrupt required by the PC architecture. Each device must implement a set of three registers that uniquely identify the device: Vendor ID (allocated by the PCI SIG), Device ID (allocated by the vendor), and Revision ID. The Class Code register identifies a programming interface (SCSI controller interface, for example), or a register-level interface (ISA DMA controller, for example). As a final example, the Device Control field specifies whether the device responds to I/O space accesses, or memory space accesses, or both, and whether the device can act as a PCI bus master. At power-up, device independent software determines what devices are present, and how much address space each device requires. The boot software then relocates PCI devices in the address space using a set of base address registers

2. Analysis of NIC

NIC is a device on PCI Bus. This high-performance adapter can plug directly into PCI Bus. In the Multi-Bus Simulator developed by us, all of the devices on PCI Bus were defined as virtual NICs. So we need to analyze the architecture and behavior of NIC.

2.1 Introduction to NIC

A NIC (network interface card) is a piece of computer hardware designed to allow computers to communicate over a computer network. It is both an OSI layer 1 (physical layer) and layer 2 (data link layer) device, as it provides physical access to a networking medium and provides a low-level addressing system through the use of MAC addresses. It allows users to connect to each other either by using cables or wireless.

Every Ethernet network card has a unique 48-bit serial number called a MAC address, which is stored in ROM carried on the card. Every computer on an Ethernet network must have a card with a unique MAC address. No two cards ever manufactured share the same address. Whereas network cards used to be expansion cards that plug into a computer bus, the low cost and ubiquity of the Ethernet standard means that most new computers have a network interface built into the motherboard. These motherboards either have Ethernet capabilities integrated into the motherboard chipset, or implemented via a low cost dedicated Ethernet chip, connected through the PCI (or the newer PCI express bus). A separate network card is not required unless multiple interfaces are needed or some other type of network is used. Newer motherboards may even have dual network (Ethernet) interfaces built-in.

2.2 Architecture of NIC

2.2.1 Basic NIC Architecture

As shown in Figure2 below, most NICs have a DMA interface unit, a medium access control (MAC) unit, memory, and control logic. In the case of a programmable NIC, the control logic is one or more programmable processors that run compiled-code hardware. The DMA unit is directed by the onboard control logic to read and write data between the local NIC memory and the host's memory. The medium access unit interacts with the control logic to receive frames into local buffer storage and to send frames from local buffer storage out onto the network. The memory is used for temporary storage of frames, buffer descriptors, and other control data.

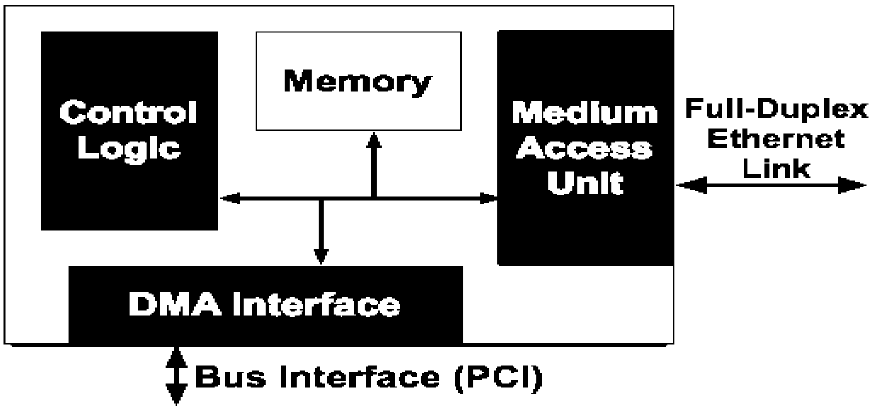


Figure2. Basic NIC Architecture

2.2.2 10 Gigabit NIC Architecture

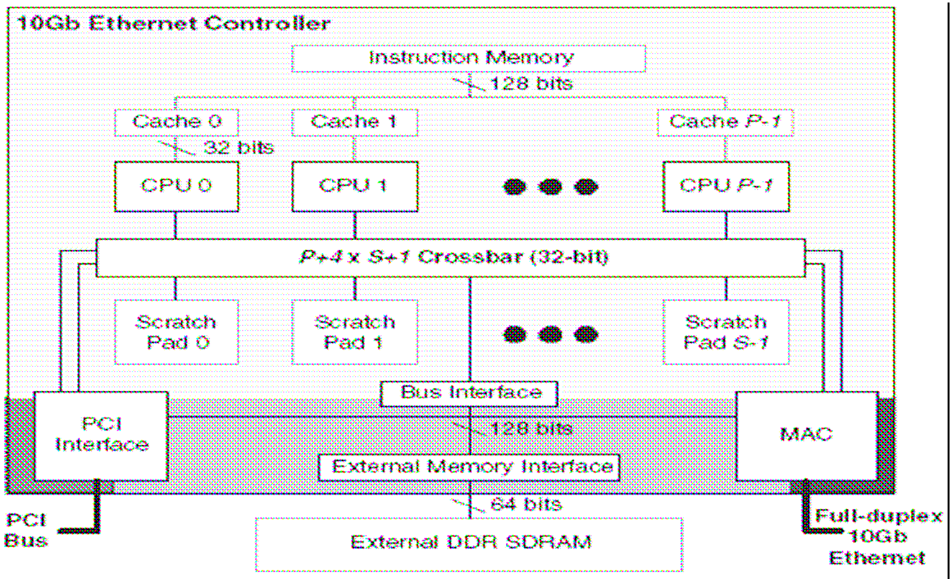


Figure3. 10 Gigabit NIC Architecture

Figure3 shows the 10 Gigabit NIC's computation and memory architecture. The controller architecture includes parallel processing cores, a partitioned memory system, and hardware assist units for performing DMA transfers across the host interconnect bus and for performing Ethernet data sends and receives according to the MAC policy. In addition to being responsible for all frame data transfers, the assist units are also involved with some control data accesses when they share information with the processors about which frame contents are to be or have been transferred. To allow stores to proceed without stalling the processor, a single store may be buffered in the MEM stage; loads requiring more than one cycle force the processor to stall. To control status flags for the event queue mechanism, each processor also implements two atomic read-modify-write operations, set and update. Set takes an index into a bit array in memory. Update examines the bit array and looks for continuous bits that have been set since the last update. Update clears the continuous set bits and returns a pointer indicating the offset where the last cleared bit was found. Hardware running on these processors can use these instructions to communicate "done" status information between computation phases and eliminate the synchronization, looping, and flag-update overheads. Instructions are stored in a single 128 KB instruction memory which feeds per-processor instruction caches. Hardware and assist control data is stored in the on-chip scratchpad, which has a capacity of 256 KB and is separated into **S** independent banks. The scratchpad is visible to all processors and hardware assist units. This provides the necessary communication between the processors and the assists as the assists read and update descriptors about the packets they process. The scratchpad also enables low-latency data sharing between processors. The processors and each of the four hardware assists connect to the scratchpads through a crossbar. There is also a crossbar connection to allow the processors to connect to the external memory interface; the assists access the external memory interface directly. The crossbar is 32 bits wide and allows one transaction to each scratchpad bank and to the external memory bus interface per cycle with round-robin arbitration for each resource. Accessing any scratchpad bank requires a latency of 2 cycles: one to request and cross the crossbar, another to access the memory and return requested data. So the processors must always stall at least one cycle for loads. If each core had its own private scratchpad, the access latency could be reduced to a single cycle by eliminating the crossbar. But, if we do that, each core may be limited to only accessing its local scratchpad or may require a much higher latency to access a remote location.

The processor cores and scratchpad banks operate at the CPU clock frequency, which could be 166 MHz in an embedded system. At this frequency, if the cores operate at 100% efficiency, cores and scratchpad banks could meet the computation and control data bandwidth. To provide enough bandwidth for bidirectional 10 GB/s data streams, the external memory bus is separated from the rest of the system because it must operate faster than the CPU cores. It is not good to force the cores to operate faster because that will waste more power. So the external memory can provide enough bandwidth for frame data. The PCI interface and MAC unit share a 128-bit bus to access the 64-bit wide external DDR SDRAM. At the same operating frequency, both the bus and the DDR SDRAM have the same peak transfer rate, since the SDRAM can transfer two 64-bit values in a single cycle. If the bus and SDRAM are able to operate at 100% efficiency, then they can achieve 40 GB/s of bandwidth. But transmit traffic cannot meet that, because it requires two transfers per frame (header and data). A 64-bit wide GDDR SDRAM provides a peak bandwidth of 64 GB/s, and is able to support 40 GB/s of bandwidth for network traffic. Since the PCI bus, the MAC interface, and the external DDR SDRAM all operate at different clock frequencies, there must be four clock areas on the chip.

2.2.3 10 Gigabit NIC Memory System

The memory system for a 10 GB/s NIC must support low-latency access to control data and high-bandwidth, high-capacity storage for frame data. Control data refers to buffer descriptors, which the processors must read and modify, and I/O descriptors that the DMA and MAC units use. For a programmable NIC, the processors create I/O descriptors which the DMA and MAC units read and process; the MAC receive unit is the exception, which creates descriptors that the processors read. To maintain high frame rates, the I/O units can't stall waiting for access to control data. Simultaneously, a 10 GB/s programmable NIC must support high-capacity storage for frame data to support offload mechanisms. Frame storage must also support at least 40 GB/s of throughput, since there are 4 simultaneous access streams (DMA read and MAC transmit for the send path, MAC receive and DMA write for the receive path) that must each maintain throughput of 10 GB/s. But no single memory structure is low-latency, has high bandwidth, and provides a large capacity.

2.2.3.1 **Multiple processing cores**

We can improve the performance of processor by using the parallelism of multiple processing cores. There are several factors that can limit the obtainable instructions per cycle (IPC) of a particular processor running network interface hardware. The major factors considered here are whether the processor issues instructions in-order or out-of-order, the number of instructions that can be issued per cycle, whether or not branch prediction is used, and the structure of the pipeline.

In the perfect pipeline, all instructions complete in a single cycle, so the only limit on IPC is that instructions that depend on each other cannot issue during the same cycle. For a typical five stage pipeline with all forwarding paths is configured, loads sequences would cause a pipeline stall and only one memory operation can issue per cycle. In addition to pipeline configurations, two branch prediction methods are configured. For the perfect branch prediction, any number of branches up to the issue width can be correctly predicted on every cycle. For no branch prediction, a branch stops any further instructions from issuing until the next cycle. For an in-order processor, it is more important to eliminate pipeline hazards than to predict branches. Conversely, for an out-of-order processor, it is more important to accurately predict branches than to eliminate pipeline hazards.

To improve the processor’s performance, the complexity may not be worth the cost for an embedded system. For example, an out-of-order processor with an issue width of two and perfect branch prediction of one branch per cycle could only achieve twice IPC performance than an in-order processor with no branch prediction and including pipeline stalls. But it has higher complexity. It would need a wide issue window that must keep track of instruction dependencies and select two instructions per cycle. It would also need a register renaming mechanism and a reorder buffer to allow instructions to execute out-of-order. And it would require a complex branch predictor to approach the performance of a perfect branch predictor. All of this complexity adds area, delay, and power dissipation to the out-of-order core.

For an area- and power-constrained embedded system, it is likely to be more efficient to use parallelism through the use of multiple processing cores, rather than by more complex cores. If the out-of-order core costs twice as much as the in-order core, it is better to use two simple in-order cores. Increasing the IPC by using the two-wide out-of-order core would require additional cost, such as predicting multiple branches per cycle or issuing four instructions per cycle. This would increase the area, delay, and power dissipation of the core. Because of that, multiple simple cores become more attractive. The use of multiple processing cores will motivate the use of simple, single-issue, in-order processing cores as the base processing element for a network interface. This will minimize the complexity, and the area, delay, and power dissipation of the system.

2.2.3.2 **Data Memory**

The combination of parallel cores and hardware assists requires a multiprocessor memory system that allows the simple processor cores to access their data and instructions with low latency to avoid pipeline stalls while also allowing frame data to be transferred at line rate. Since the frame data is not accessed by the processing engines of the network interface, it can be stored in a high-bandwidth off-chip memory. However, the instructions and frame metadata accessed by the processor must be stored in a low-latency, random access memory. Because instructions are read-only, accessed only by the processors and have a small working set, per-processor instruction caches are a natural solution for low-latency instruction access. The frame metadata also has a small working set, fitting entirely in 100 KB. However, the metadata must be read and written by both the processors and the assists. Per-processor coherent caches are a well-known method for providing low-latency data access in multiprocessor systems. Caches are also transparent to could have its own private cache for frame metadata, with frame data set to bypass the caches in order to avoid pollution.

But these structures also have several disadvantages. Caches waste space by requiring tag arrays in addition to the data arrays that hold the actual content. Caching also wastes space by replicating the same widely-read data across the private caches of several processors. All coherence schemes add complexity and resource occupancy at the controller. A major problem with a hierarchy design such as this stems from the requirement that control data must be coherent. That is, data written by the processors (such as buffer descriptors, I/O descriptors, and so forth) must be observed by the I/O units, and vice versa. A write back policy would maintain coherence but would overwhelm the main memory with a random pattern of write backs such that the DRAM could not satisfy the streaming frame data. Conversely, memory trace analysis shows that in a cache-coherent scheme, coherence messages overwhelm the main memory bus so that, again, the DRAM cannot satisfy the streaming frame data.

A possible solution used in embedded systems is the use of a program-controlled scratchpad memory. This is a small region of on-chip memory dedicated for low-latency accesses, but all contents must be explicitly managed by the program. Such a scratchpad memory would provide enough control data bandwidth. But simultaneous accesses from multiple processors would incur queuing delays. These delays can be avoided by splitting the scratchpad into multiple banks, providing excess bandwidth to reduce latency. A banked scratchpad requires an interconnection network between the processors and assists on one side and the scratchpads on the other. Such networks yield a tradeoff between area and latency, with a crossbar requiring at least an extra cycle of latency. On the other hand, the scratchpad avoids the waste, complexity, and replication of caches and coherence. Since the processors do not need to access frame data, frame data does not need to be stored in a low-latency memory structure. Furthermore, frame data is always accessed as four 10 GB/s sequential streams with each stream coming from one assist unit. Current SDRAM can provide sufficient bandwidth for all of these streams. By providing enough buffering for frames in each assist, data can be transferred between the assists and the SDRAM up to 1518 bytes at a time. These transfers are to consecutive memory locations, so using an arbitration scheme that allows the assists to sustain such bursts will incur very few row activations in the SDRAM and allow peak bandwidth to be achieved during these bursts.

2.2.4 Send and receive of NIC

NICs send and receive frames between the networks and host operating system. Sending and receiving frames happens in a series of steps completed by the host and NIC.

The host operating system of a network server uses the network interface to send and receive packets. The operating system stores and retrieves data directly to or from the main memory, and the NIC transfers this data to or from its own local transmit and receive buffers. Sending and receiving data is handled cooperatively by the NIC and the device driver in the operating system, which notify each other when data is ready to be sent or has just been received.

The host OS maintains a series of buffers used for frame headers and contents. The OS also maintains a queue or ring of buffer descriptors. Each buffer descriptor indicates where in host memory the buffer resides and how big the buffer is. Buffer descriptors are the unit of transaction between the OS and NIC - when the OS needs to indicate that frames are ready to be sent, the NIC must fetch and process the buffer descriptors that describe those pending frames. Likewise, the OS indicates that free buffers are available to write received frames into by making buffer descriptors available to the NIC that point to free space in host memory. After received frames arrive, the NIC sends back completed buffer descriptors that indicate the size of the received frames.

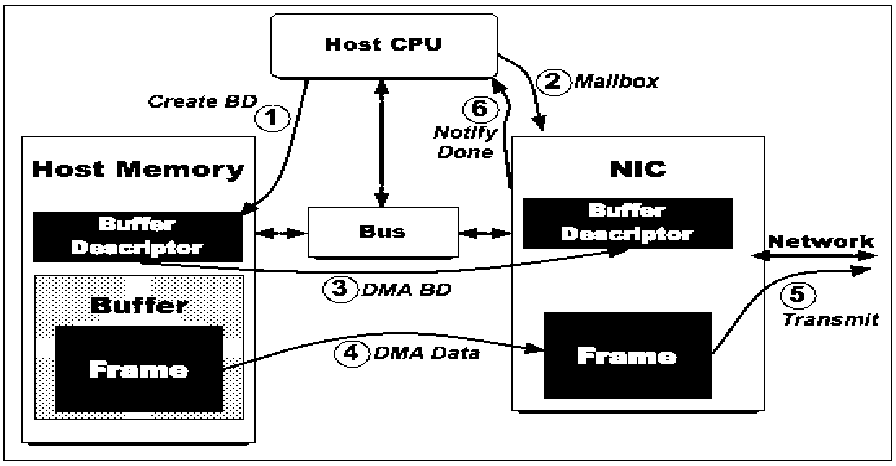


Figure4. Steps involved in sending a packet

Sending a packet requires the steps shown in Figure4.

Step1, the device driver first creates a buffer descriptor, which contains the starting memory address and length of the packet that is to be sent, along with additional flags to specify options or commands. If a packet consists of multiple non-contiguous regions of memory, the device driver creates multiple buffer descriptors.

Step2, the device driver then writes to a memory-mapped register on the NIC with information about the new buffer descriptors.

Step3, the NIC initiates one or more direct memory access (DMA) transfers to retrieve the descriptors.

Step4, the NIC initiates one or more DMA transfers to move the actual packet data from the main memory into it's transmit buffer using the address and length information in the buffer descriptors.

Step5, after the packet is transferred, the NIC sends the packet out onto the network through its medium access control (MAC) unit. The MAC unit is responsible for implementing the link-level protocol for the underlying network such as Ethernet.

Step6, the NIC informs the device driver that the descriptor has been processed, possibly by interrupting the CPU.

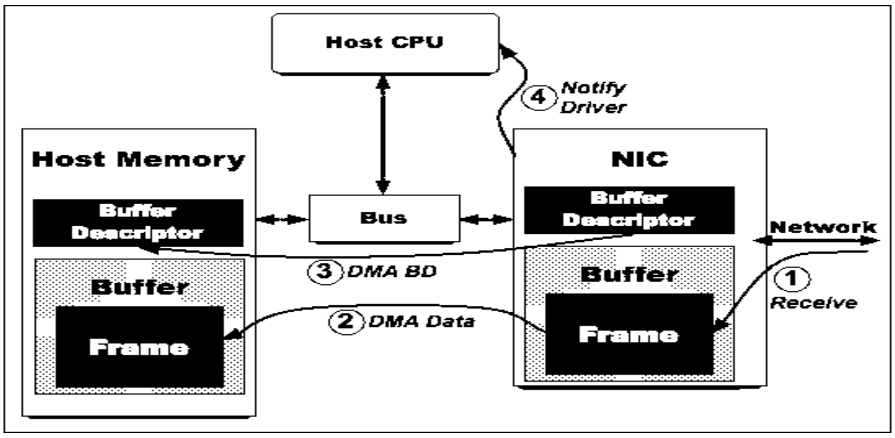


Figure5. Steps involved in receiving a packet

Receiving packets is analogous to sending them, but the device driver must also pre-allocate a pool of main-memory buffers for arriving packets. Because the system cannot anticipate when packets will arrive or what their size will be, the device driver continually allocates free buffers and notifies the NIC of buffer availability using buffer descriptors. The notification and buffer-descriptor retrieval processes happen just as in the send case, following steps 1 through 3 of Figure4.

Figure5 depicts the steps for receiving a packet from the network into pre-allocated receive buffers.

Step1, a packet arriving over the network is received by the MAC unit and stored in the NIC's local receive buffer.

Step2, the NIC initiates a DMA transfer of the packet into a pre-allocated main memory buffer.

Step3, the NIC produces a buffer descriptor with the resulting address and length of the received packet and initiates a DMA transfer of the descriptor to the main memory, where it can be accessed by the device driver.

Step4, the NIC notifies the device driver about the new packet and descriptor, typically through an interrupt. The device driver may then check the numbers of unused receive buffers in the main memory and refill the pool for future packets.

In the sending and receiving cases, NIC processing breaks up into several steps, some of which have very long latencies (e.g., waiting for DMAs to complete, waiting for frames to arrive, and so on). To tolerate these latencies, NIC hardware uses an event model of computation. Events are typically associated with the completion of one of the NIC processing steps that require further processing. In the event model, the processors wait for events to arrive and then let specific event handlers to process the newly-arrived events. These handlers may en-queue new long-latency I/O operations which will trigger more events. After each event handling functions return, the processors resume waiting for other events. Hence, the processors can overlap processing of intermediate steps of a frame with latencies associated with another frame.

To send and receive frames, a programmable Ethernet controller would need one or more processing cores and several specialized hardware assist units that efficiently transfer data to and from the local interconnect and the network. The Fetch Send BD and Fetch Receive BD tasks fetch buffer descriptors from the main memory that specify the location of frames to be sent or of pre-allocated receive buffers (step 3 of Figure4 and Figure5). Send Frame and Receive Frame implement steps 4-6 of Figure4 and 1-4 of Figure5, respectively. Fetch Send BD and Fetch Receive BD transfer multiple buffer descriptors through a single DMA. And each sent frame typically requires two buffer descriptors because the frame consists of two discontinuous memory regions, one for the frame headers and one for the payload.

A full-duplex 10 GB/s link can deliver maximum-sized 1518-byte frames at the rate of 812,744 frames per second in each direction. Each sent or received frame must be first stored into the local memory of the NIC and then read from the memory. For example, to send a frame, the NIC first transfers the frame from the main memory into the local memory, and then the MAC unit reads the frame from the local memory. Thus, sending and receiving maximum-sized frames at 10 GB/s require 39.5 GB/s of data bandwidth. This is slightly less than the overall link bandwidth (2 * 2*10 GB/s) because data cannot be sent during the Ethernet inter-frame gap.

2.2.5 Hardware Parallelism of NIC

As discussed in the previous section, an efficient programmable 10 GB/s network interface must use parallel computation cores, per-processor instruction caches, scratchpad memories for control data, and high-bandwidth SDRAM for frame contents. In order to successfully utilize such architecture, however, the Ethernet hardware running on the interface must be parallelized appropriately. As depicted in Figures 4 and 5, network interfaces experience significant latencies when carrying out the steps necessary to send and receive frames. Most of these latencies stem from requests to access host memory via DMA. To tolerate the long latencies of interactions with the host, previous NIC hardware uses an event-based processing model in which the steps outlined in Figures 4 and 5 are mapped to separate events. For the triggering of an event, the hardware runs a specific event handler function for that type of event. Events may be triggered by hardware completion notifications (e.g., packet arrival, DMA completion) or by other event handler functions that wish to trigger a software event.

2.2.5.1 Previous Hardware Parallelism

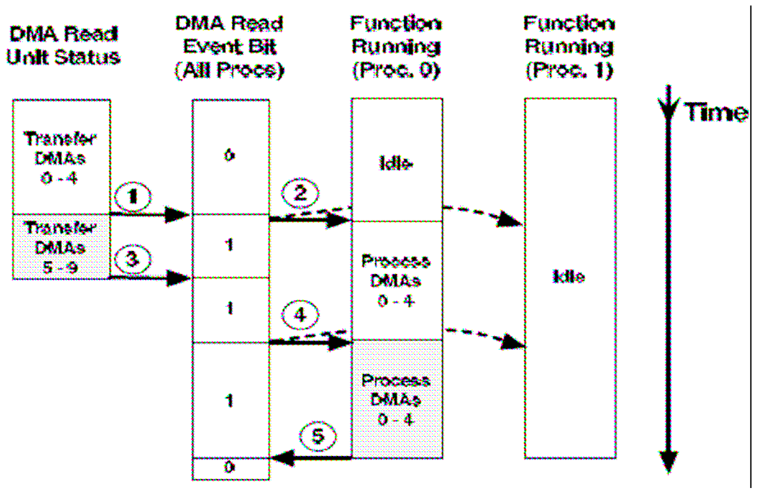


Figure6. Previous hardware parallelism (with an event register)

Event notification mechanism runs different event handlers concurrently. The Ethernet controller uses a hardware-controlled event register to indicate which types of events are pending. An event register is a bit vector in which each bit corresponds to a type of event. A set bit indicates that there is one or more events of that type that need processing. Figure 6 shows how the parallel hardware uses an event register to detect a DMA read event and dispatch the appropriate event handler.

Step1, the DMA hardware indicates that some DMAs have completed by setting the global DMA read event bit in every processors' event register.

Step2, processor 0 detects this bit becoming set and dispatches the Process DMAs event handler.

Step3, when DMAs 5-9 complete at the DMA read hardware, the hardware again attempts to set the DMA read event bit.

Step4, processor 0 marks its completed progress as it finishes processing DMAs 0-4. Since DMAs 5-9 are still outstanding, the DMA read event bit is still set. At this time, either processor 0 or processor 1 could begin executing the Process DMAs handler.

Step5, since no more DMAs are outstanding, processor 0 again marks its progress and clears the DMA read event bit. Even though DMAs become ready for processing at step 3 and processor 1 is idle, no processor can begin working on DMAs 5-9 until processor 0 finishes working on DMAs 0-4.

This weak point is the result of an event register mechanism. The event register only indicates that DMAs are ready, but it does not indicate which DMAs are ready. So, if a processor is engaged in handling a specific type of event, no other processor can handle that same type of event without significant overhead. If we divide events into work units, and the processors and hardware assists would have to cooperate in some manner to decide when to turn event bits on and off. This method will improve the idle time between steps 3 and 4 so long as the handlers are well-balanced across the processors. However, the event handlers cannot be balanced across many processors.

2.2.5.2 Modern Hardware Parallelism

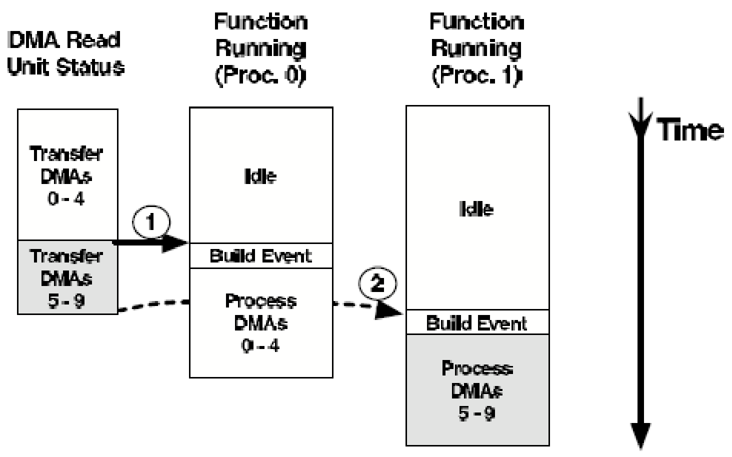


Figure7. Modern hardware parallelism (with a distributed event queue)

In Figure 6, the processing of DMAs 5-9 does not depend on the processing of DMAs 0-4. Rather than dividing work according to type and executing different types in parallel, modern parallel method divides work into bundles of work units that need a certain type of processing. Once divided, these work units (described by an event data structure) can be executed in parallel, regardless of the type of processing required. This modern parallel organization enables higher levels of concurrency but requires some additional overhead to build event data structures and maintain frame ordering.

Figure 7 illustrates how a modern parallel hardware processes the same sequence of DMAs previously illustrated in Figure 6. As DMAs complete, the DMA hardware updates a pointer that marks its progress.

Step1, processor 0 inspects this pointer, builds an event structure for DMAs 0 through 4, and executes the Process DMAs handler.

Step2, processor 1 notices the progress that indicates DMAs 5 through 9 have completed, builds an event structure for DMAs 5 through 9, and executes the Process DMAs handler.

Unlike the previous parallel hardware in Figure 6, two instances of the Process DMAs handler can run concurrently. As a result, idle time only occurs when there is no other work to be done. As indicated by Figure 7, a modern parallel hardware must inspect several different hardware-maintained pointers to detect events. Such hardware must maintain a queue of event structures to facilitate software-raised events and retries. Software-raised events signal the need for more processing in another NIC-processing step, while retries are necessary if an event handler temporarily exits local NIC resources.

Frames may complete their processing out-of-order with respect to their arrival order. In-order frame delivery must be ensured to avoid the performance degradation associated with out-of-order TCP packet delivery. To facilitate this, the hardware maintains several status buffers where the intermediate results of network interface processing may be written. The hardware inspects the final-stage results for a “done” status and submits all subsequent, consecutive frames. The task of submitting a frame may not be run concurrently, but submitting a frame only requires a pointer update. As frames progress from one step of processing to the next, status flags become set that indicate one stage of processing has completed and another is ready to begin. However, hardware pointer updates require that a consecutive range of frames is ready. To determine if a range is ready and update the pointer, the hardware processors must synchronize, check for consecutive set flags, clear the flags, update pointers and then finally release synchronization. These synchronized, looping memory accesses represent an important source of overhead.

2.3 Simulating and testing data of virtual NIC

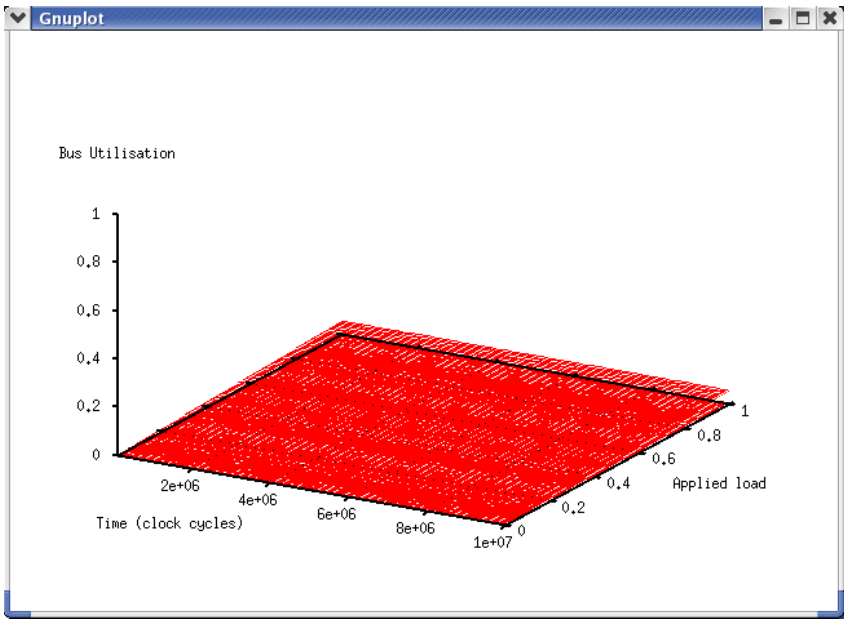
Our simulator is developed by C/C++/G++ and top-down method. It works or executes under Redhat Linux platform. As for the software architecture, It is consists of three sections: head file which mainly focuses on the definition of parameters and declaration of functions, main program which shows us the definition of relative functions and main function, and plot program which call relative Gnuplot commands to draw corresponding diagrams according to users’ requirements.

In our simulator, virtual NICs were defined as the devices on PCI Bus. We can get a lot of data information after running our simulator. For the analysis of data, 3 plotting programs of Gnuplot were developed. They are data rate plotting program, throughput plotting program and transfer time histogram plotting program. (For codes of the 3 plotting programs, see Appendix).

2.3.1 Testing data and plotting

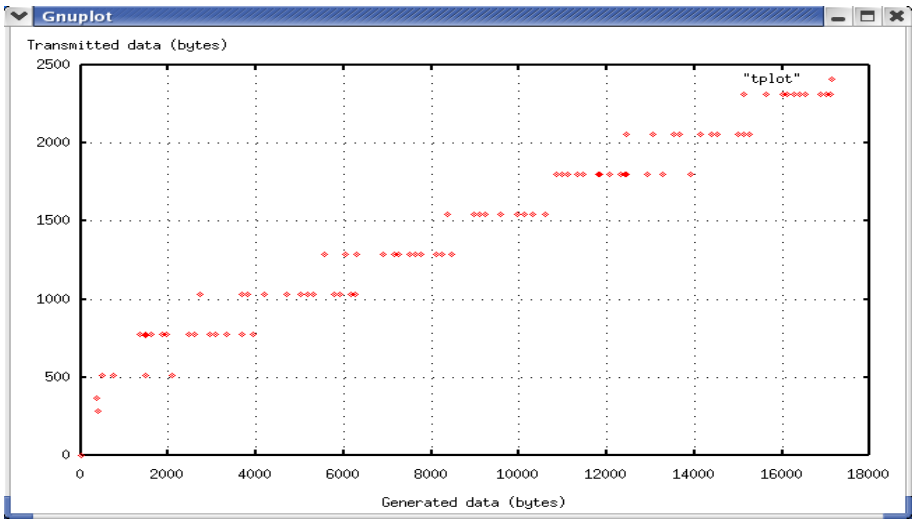
2.3.1.1 Single bus simulator

```
# Data rate plot
# ==============
# Sample slot width (clock cycles): 10000
# No. of masters:                    4
# Bus frequency (MHz):               66.0
# Size of data objects (bytes):      8
# Arbitration scheme:                Fixed
#
# BUS0 devices attached as masters:
#      R/W   Prio   Block size    Data rate    Max. WS   S/D   LT
#   1   R     1        129         230000.0       2       D     4
#   2   R     2        256         120000.0       2       D     2
#   3   W     3        222         330000.0       2       D     3
#   4   W     4        129         450000.0       2       D     2
#      Time          Load       Bus utilization
       10000       0.010000     0.000000
       20000       0.010000     0.000000
       30000       0.010000     0.000000
       40000       0.010000     0.000000
       50000       0.010000     0.000000
       .......      ......      .......
       60000       0.010000     0.000000
       70000       0.010000     0.000000
       80000       0.010000     0.000000
       .......      ......      .......
      100000       0.010000     0.000000
      110000       0.010000     0.000000
```



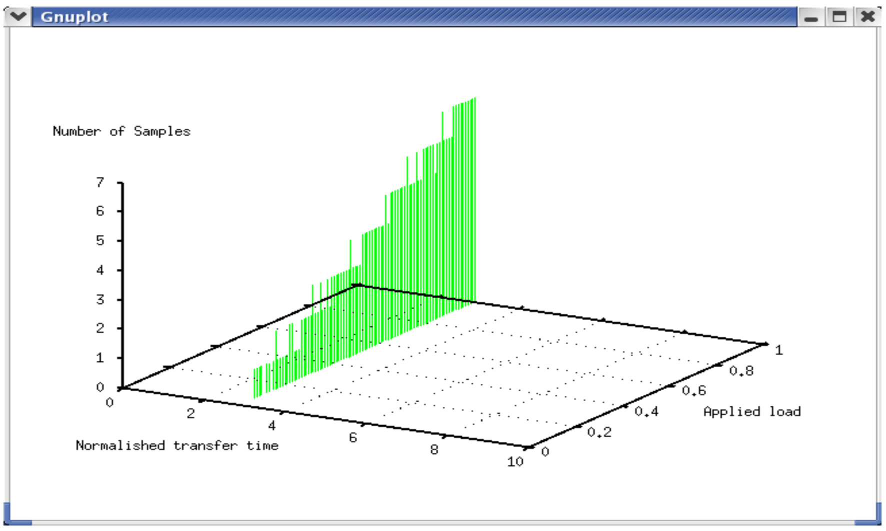
Plot1.Data rate (single bus)

```
# Throughput plot
# ===============
# No. of masters:                     4
# Bus frequency (MHz):                 66.0
# Size of data objects (bytes):     8
# Arbitration scheme:                Fixed
#
# BUS0 devices attached as masters:
#      R/W   Prio    Block size      Data rate     Max. WS   S/D    LT
#   1   R      1          129           230000.0          2        D         4
#   2   R      2          256           120000.0          2        D         2
#   3   W      3           222           330000.0          2        D         3
#   4   W      4           129           450000.0          2        D         2
# Generated data          Transmitted data
                  0                        0
                351                      368
                480                      512
                387                      280
                ……                      ……
                738                      512
              1345                      776
              1474                      776
              1474                      768
                ……                      ……
              1603                      776
              1476                      512
              1954                      776
              1861                      776
              2085                      512
              2470                      776
```



Plot2. Throughput plot (single bus)

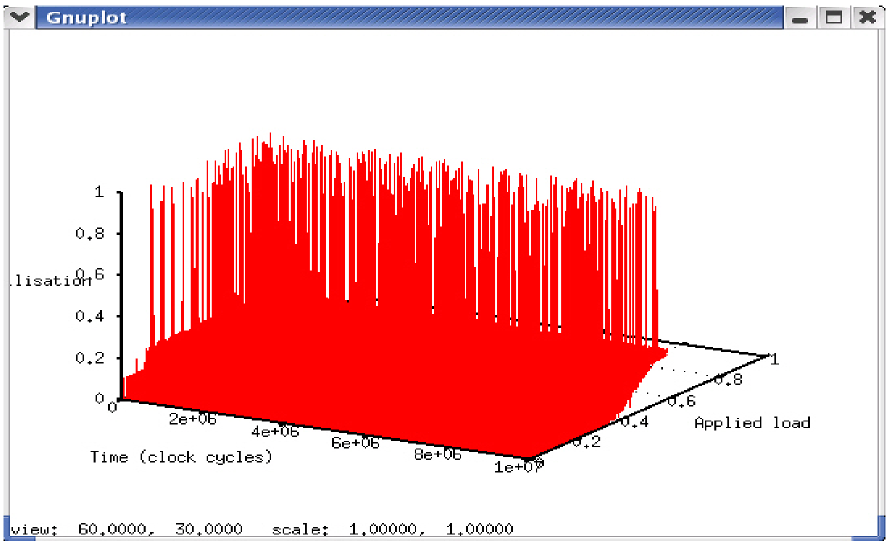
```
# Transfer time histogram
# ======================
# No. of masters:                  4
# Bus frequency (MHz):              66.0
# Size of data objects (bytes):    8
# Arbitration scheme:              Fixed
#
# BUS0 devices attached as masters:
#     R/W   Prio   Block size    Data rate    Max. WS   S/D    LT
#   1   R     1        129       230000.0       2       D       4
#   2   R     2        256       120000.0       2       D       2
#   3   W     3        222       330000.0       2       D       3
#   4   W     4        129       450000.0       2       D       2
#Transfer time     Load      Samples
   0.100000      0.010000        0       0       0       0
   0.200000      0.010000        0       0       0       0
   0.300000      0.010000        0       0       0       0
   0.400000      0.010000        0       0       0       0
   0.500000      0.010000        0       0       0       0
   ……….          ………          …       …       ..      ..

   1.000000      0.010000        0       0       0       0
   1.100000      0.010000        0       0       0       0
   1.200000      0.010000        0       0       0       0
   1.300000      0.010000        0       0       0       0
   1.400000      0.010000        0       0       0       0
   ……….          ………          …       …       ..      ..
```



Plot3. Transfer time histogram (single bus)

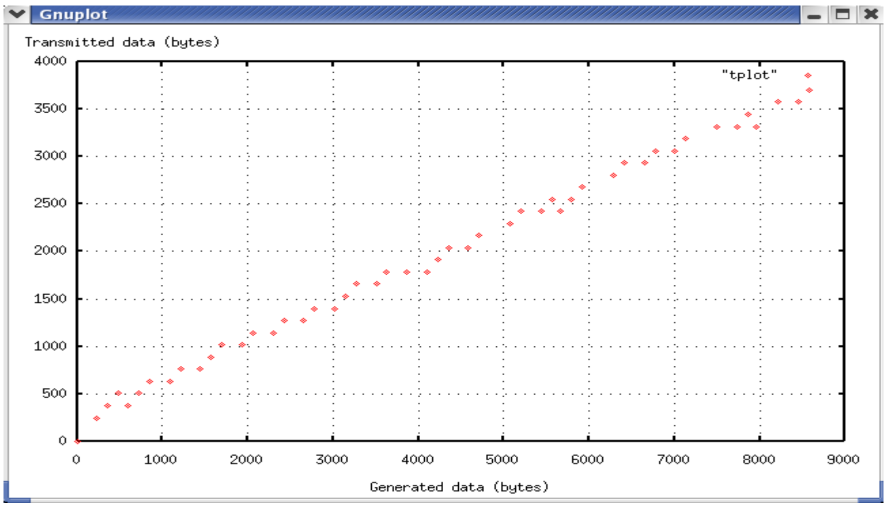
2.3.1.2 Two bus simulator

```
# Data rate plot
# ==============
# Sample slot width (clock cycles): 10000
# No. of masters:                      2
# Bus frequency (MHz):                  33.0
# Size of data objects (bytes):       4
# Arbitration scheme:                  Fixed
#
# BUS0 devices attached as masters:
#    R/W/B   Prio    Block size      Data rate     Max. WS    S/D      LT
#   1   R       1          239             230000.0          2        D         2
#   2   W       2           129             440000.0          2        D         2
#       Time               Load         Bus utilization
#
# BUS1 devices attached as masters:
#    R/W/B   Prio    Block size      Data rate     Max. WS    S/D     LT
#   1   R       3          128             230000.0          2        D         3
#   2   W       4           128             310000.0          2        D         3
#
# Simulation time (clock cycles):     100000
#       Time               Load         Bus utilization
                1          0.010000       0.000000
                2          0.010000       0.000000
                4          0.010000       0.000000
               …          ………            ………
               18          0.010000       0.000000
               19          0.010000       0.000000
               20          0.010000       0.000000
               21          0.010000       0.000000
               22          0.010000       0.000000
               …          ………            ………
               35          0.010000       0.000000
               36          0.010000       0.000000
               …          ………            ………
```



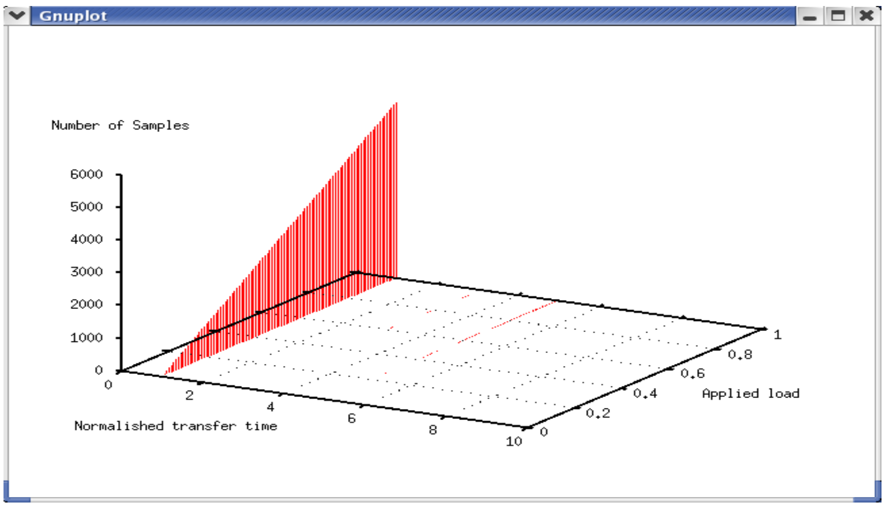
Plot4. Data rate (two buses)

```
# Throughput plot
# ===============
# No. of masters:                      2
# Bus frequency (MHz):                  33.0
# Size of data objects (bytes):        4
# Arbitration scheme:                  Fixed
#
# BUS0 devices attached as masters:
#    R/W/B   Prio    Block size     Data rate    Max. WS   S/D     LT
#   1   R       1         239          230000.0         2       D        2
#   2   W        2          129          440000.0         2       D        2
# Generated data   Transmitted data
#
# BUS1 devices attached as masters:
#    R/W/B   Prio    Block size     Data rate    Max. WS   S/D    LT
#   1   R       3         128          230000.0         2       D        3
#   2   W        4          128          310000.0         2       D        3
#
# Simulation time (clock cycles):     100000
# Generated data          Transmitted data
              0                      0
            501                    368
          …….                     ……
            5523                   3210
            6877                   3968
            8225                   4023
            …….                   ……
```



Plot5. Throughput plot (two buses)

```
# Transfer time histogram
# ======================
# No. of masters:                        2
# Bus frequency (MHz):                    33.0
# Size of data objects (bytes):     4
# Arbitration scheme:                   Fixed
#
# BUS0 devices attached as masters:
#      R/W/B   Prio     Block size       Data rate     Max. WS    S/D      LT
#    1    R         1            239             230000.0           2          D          2
#    2    W         2             129             440000.0           2          D          2
#Transfer time        Load         Samples
#
# BUS1 devices attached as masters:
#      R/W/B   Prio     Block size       Data rate     Max. WS    S/D     LT
#    1    R         3            128             230000.0           2          D          3
#    2    W         4             128             310000.0           2          D          3
#
# Simulation time (clock cycles):      100000
#Transfer time        Load         Samples
    0.100000          0.010000            0               0
    0.200000          0.010000            0               0
    0.300000          0.010000            0               0
    ………..            ……….             …              …
    1.300000          0.010000            0               0
    1.400000          0.010000            0               0
    1.500000          0.010000            0               0
    ………..            ……….             …              …
```



Plot6. Transfer time histogram (two buses)

2.3.1.3 Data collection of Multi-Bus Simulator

```
# Throughput plot
# ===============
# No. of masters:                        5
# Bus frequency (MHz):                   33.0
# Size of data objects (bytes):      4
# Arbitration scheme:                    Fixed
#
# BUS0 devices attached as masters:
#      R/W/B    Prio     Block size        Data rate     Max. WS    S/D      LT
#    1    R        1            128          230000.0           2        D          3
#    2    R        2            128          210000.0           2        D          3
#    3    W        3            243          250000.0           1        D          2
#    4    W        4            432          230000.0           4        D          3
#    5    R        5            239          340000.0           2        D          3
# Generated data    Transmitted data
#
# BUS1 devices attached as masters:
#      R/W/B    Prio     Block size        Data rate     Max. WS    S/D     LT
#    1    R        6            129                2.0           0        D          2
#    2    W        7            128          120000.0           2        D          4
# Simulation time (clock cycles):     1000000
# Generated data    Transmitted data
#
# BUS2 devices attached as masters:
#      R/W/B    Prio     Block size        Data rate     Max. WS    S/D     LT
#    1    R        8            145          350000.0           1        D          2
#    2    R        9        120000                2.0           2        D          2
#
```

```
# Simulation time (clock cycles):      1000000
# Generated data    Transmitted data
#
# BUS3 devices attached as masters:
………………….
# Simulation time (clock cycles):      1000000
# Generated data    Transmitted data
#
# BUS4 devices attached as masters:
#     R/W/B   Prio    Block size       Data rate     Max. WS   S/D     LT
#    1   R       10        240000              230000.0          2          D          3
#    2   R       13            230              230000.0          2          D          2
#
# Simulation time (clock cycles):      1000000
# Generated data    Transmitted data
                 0                          0
                 0                          0
               ……                       ….
```

```
# Transfer time histogram
# =====================
# No. of masters:                              5
# Bus frequency (MHz):                          33.0
# Size of data objects (bytes):          4
# Arbitration scheme:                         Fixed
#
# BUS0 devices attached as masters:
#     R/W/B   Prio    Block size       Data rate     Max. WS   S/D      LT
#    1   R        1            128              230000.0          2          D          3
#    2   R        2            128              210000.0          2          D          3
#    3   W        3             243              250000.0          1          D          2
#    4   W        4             432              230000.0          4          D          3
#    5   R        5            239              340000.0          2          D          3
#Transfer time       Load          Samples
#
# BUS1 devices attached as masters:
#     R/W/B   Prio    Block size       Data rate     Max. WS   S/D     LT
#    1   R        6            129                      2.0          0          D          2
#    2   W        7             128              120000.0          2          D          4
#
# Simulation time (clock cycles):      1000000
#Transfer time       Load          Samples
# Simulation time (clock cycles):      1000000
```

```
#Transfer time      Load        Samples
#
# BUS2 devices attached as masters:
#     R/W/B   Prio    Block size      Data rate     Max. WS    S/D     LT
#   1   R       8           145           350000.0         1        D         2
#   2   R       9       120000                  2.0         2        D         2
#
# Simulation time (clock cycles):     1000000
#Transfer time      Load        Samples
#
# BUS3 devices attached as masters:
#     R/W/B   Prio    Block size      Data rate     Max. WS    S/D     LT
#
# Simulation time (clock cycles):     1000000
#Transfer time      Load        Samples
#
# BUS4 devices attached as masters:
#     R/W/B   Prio    Block size      Data rate     Max. WS    S/D     LT
#   1   R      10       240000            230000.0         2        D         3
#   2   R      13           230           230000.0         2        D         2
#
# Simulation time (clock cycles):     1000000
#Transfer time      Load        Samples
    0.100000        0.010000          0           0
    0.200000        0.010000          0           0
    0.300000        0.010000          0           0
    0.400000        0.010000          0           0
    0.500000        0.010000          0           0
    0.600000        0.010000          0           0
    0.700000        0.010000          0           0
    0.800000        0.010000          0           0
    0.900000        0.010000          0           0
    1.000000        0.010000          0           0
    ………..          …………           ….          ….
```

For Multi-bus simulator, we can not plot with relative data generated, because there are some bugs existing in the source code. Some simple analysis of plots shows below:

Type of bus	Throughput plot	Data rate plot	Histogram
Single	Theoretically, the generated data and the transmitted data should be the same, but there are some deviations in the actual plotting results. In most cases, both near 2:1 ratio.	This 3D plot is a relationship description of time, utilization and applied load. We can find that the basic bus utilization sometimes is zero, sometimes is not aero. When the volume of data increases, there will be obvious changes. The utilization of PCI bus is increased.	When the volume of data increases, the sampling results become ladderlike. And the PCI bus becomes busier.
Multi	The volume of data becomes greater, and the plot becomes complex slightly.	The volume of data becomes greater, and the plot becomes complex slightly.	The volume of data becomes greater, and the plot becomes complex slightly.

2.3.2 NIC Memory Control (MC) model's algorithm

In our PCI Multi-Bus Simulator, the virtual NIC is just a simple simulation which doesn't include the algorithm of Memory Control. So I developed my own algorithm to control the memory system of virtual NIC.

```
//Simple introduction to the NIC Memory Control (MC) model's algorithm
//This description refers to initial (size, rate), get_request (), grant (), grant_end ()
//In this algorithm:
// (1) suppose that PCI Bus access priority is always higher than extranet access priority.
// (2) define extra simple control valuable such as mn/st except the NIC device's initial parameters.
// (3) mn or st may use value: 00/01/10/11 separately to make NIC do corresponding operation
according to these values.
// (4) the NIC's operation will do R, W, or R/W (B) according to the original interactive input.

//The relative algorithm description goes as follow:
//==============================================================
NIC_initial (size, rate); //when the program start to execute, it will finish this initialization.

grant ();//PCI arbitrator give the PCI bus right to the current NIC, this function will record the
        //current system time.

get_request () {
 When the NIC get the PCI bus control right {
        (1) Record the current system time
        (2) Check current buffer size: how many data packet in the buffer
        (3) Get the latest former system time and data packet size in/out the buffer to calculate the
            value of packet change--->then figure out the packet rate.
        (4) Do relative data transaction operation by call process_request ();
        (5) Check if there still exist other requests to the PCI Bus for data transaction and record
            the current request status.
  };
};

grant_end () {
      After finishing any request operation
      Record the finish time
      Release PCI control right to arbitrator for other device use.
}

process_request () {
        //In this section, the NIC will deal with related data transaction
        //between PCI Bus and NIC buffer and then between extranet and NIC buffer
        //the NIC MC always responds to the PCI bus direction's transaction firstly
        //and then extranet (Internet) transaction, because in MC of NIC, there is
        //a internal arbitrator for the NIC internal arbitration

      if (NIC_buffer != empty){
              (1) MC will let NIC buffer to communicate with PCI bus for data transaction: read
                  or write or read/write in a limited time_quantum
              (2) As for the operation, it has been defined in the program's initialization which is
                  implemented in the interactive GUI
              (3) The operation will execute by using Swtich () procedure which include 4 phase
                  ADDRESS/IDEL/WAIT/DATA
              (4) After finishing the data transaction between NIC buffer and PCI bus in a fixed
                  time quantum, the MC will let NIC buffer communicate with extranet for
                  relative data transactions such as read, write, or Both of them.
              (5) After the transaction operations between NIC buffer and Internet, the NIC's MC
                  will release the bus control right to PCI bus's other NIC or devices.

                  Notice: all data transaction must be operated in a time_quantum
        };
      if (NIC_buffer == empty){
              (1) MC will let NIC buffer to receive data from PCI bus if this NIC has read request
                  in a time_quantum, or else, MC will check NIC if it want to read data from
                  Internet to NIC buffer.
              (2) If NIC does not want to receive data from Internet to fill the empty NIC buffer,
                  the MC will transfer its control right to PCI bus's arbitrator for other devices'
```

```
                competition/use.
        };
};

//if the NIC wants to do both R/W data transactions, we use small algorithm to implement it:

int timer = 1;    //define a timer and the initial value is 1;

while (timer != 0){
        if (time<=time_quantum){
            if (timer%2 == 1){
                do Read operation;
                timer++;
              }
            else{
                do Write operation;
                timer++;
                };
            };
};
```

References

[1] PCI Local Bus Specification, revision 3.0, August 12, 2002

[2] PCI Local Bus Specification, revision 3.0, February 3, 2004

[3] Can User-Level Protocols Take Advantage of Multi-CPU NICs? By P. Shivam, P. Wyckoff, and D. Panda.

[4] Alteon Networks. Tigon/PCI Ethernet Controller, Aug. 1997. Revision 1.04.

[5] Alteon WebSystems. Gigabit Ethernet/PCI Network Interface Card: Host/NIC Software Interface Definition. July 1999. Revision 12.4.13.

[6] End-to-End Performance of 10 Gigabit Ethernet on Commodity Systems by J. Hurwitz and W. Feng. *IEEE Micro*, Jan. /Feb. 2004.

[7] Parallelization Strategies for Network Interface Firmware by Michael Brogioli, Paul Willmann, Scott Rixner.

[8] Alteon WebSystems, Inc. Gigabit Ethernet/PCI Network Interface Card. Revision 12.4.13

[9] Official documentation of Gnuplot by Thomas Williams & Colin Kelley. Version 4.2

[10] An Efficient Programmable 10 Gigabit Ethernet Network Interface Card by P. Willmann, H. Kim, V. S. Pai, and S. Rixner.

[11] Intel PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer’s Manual

[12] http://www.ece.rice.edu/~willmann/teng_nics_hownicswork.html

[13] Design and Implementation of PCI Multi-Bus Simulator by Xiaoyong Liao

[14] Simulation of Cache Memory Systems on Symmetric Multiprocessors with Educational Purposes by M. A. Vega Rodr´ıguez, J. M. S´anchez P´erez, R. Mart´ın de la Monta˜na, and F. A. Zarallo Gallardo.

[15] A Liberty-based Simulator for Programmable Network Interface Architectures by P.Willmann, M. Brogioli, and V. S. Pai. Spinach. July 2004.

Appendix

Data rate.gnp

```
set terminal x11
set output
set noclip points
set clip one
set noclip two
set border
set boxwidth
set dummy x,y
……….
set samples 100,100
set isosamples 10,10
set surface
set nocontour
set clabel
set nohidden3d
set cntrparam order 4
set cntrparam linear
set cntrparam levels auto 5
set cntrparam point 5
set size 1,1
set data style lines
set function style lines
set tics out
set ticslevel 0.0
set xtics 0,2500000
set ytics 0.0,0.2
set ztics
set grid
set xzeroaxis
set yzeroaxis
………….
set ylabel "Applied load" 0,0
set zlabel "Bus Using" 0,0
set autoscale r
set autoscale t
set autoscale xy
set yrange [0.0 : 1.0]
set autoscale z
set zrange [0.0 : 1.0]
set zero 1e-08
```

Throughput.gnp

```
set terminal x11
set output
set noclip points
set clip one
set noclip two
set border
set boxwidth
set dummy x,y
……..
set nopolar
set angles radians
set noparametric
…….
set nocontour
set clabel
set nohidden3d
set cntrparam order 4
set cntrparam linear
set cntrparam levels auto 5
set cntrparam points 5
set size 1,1
set data style points
set function style lines
set xzeroaxis
set yzeroaxis
set tics in
set ticslevel 0.5
set xtics
set ytics
set ztics
……..
set autoscale r
set autoscale t
set autoscale xy
set autoscale z
set zero 1e-08
```

Transfer time histogram.gnp

```
set terminal x11
set output
set noclip points
set clip one
set noclip two
set border
set boxwidth
……….
set nologscale
set offsets 0,0,0,0
set nopolar
set angles radians
set parametric
………..
set nohidden3d
set cntrparam order 4
set cntrparam linear
set cntrparam points 5
set size 1,1
set data style impulses
set function style impulses
set tics out
set ticslevel 0.0
set xtics 0,2
set ytics 0.0,0.2
set ztics
set grid
set xzeroaxis
set yzeroaxis
set title "" 0,0
………..
set ylabel "Applied load" 0,0
set zlabel "Number of Samples" 0,0
set autoscale r
set autoscale t
set autoscale xy
set yrange [0.0 : 1.0]
set xrange [0.0 : 10.0]
set autoscale z
set zero 1e-08
```