A

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE HONORS DEGREE OF

BACHELOR OF SCIENCE IN PHYSICS

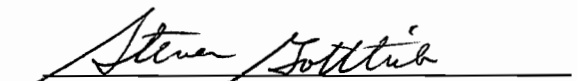
The IP over IDE Project: A New Tool for Lattice QCD

Status Report

By Zach Etienne

Department of Physics
Indiana University

4/21/03

Supervisor: 
Prof. Steven Gottlieb

Co-signer: 
Prof. Rick Van Kooten

Abstract

Due to its cost effectiveness and potential performance, IP† over IDE‡ may become the networking scheme of choice for many Beowulf clusters in the future. This paper includes a comprehensive status report which discusses in detail the motivations behind an IP over IDE networking implementation, outlines progress in the IP over IDE project from its inception (Oct. 2002) to the present, provides benchmarks for comparison to other networking schemes (including IP over SCSI# and IP over fast Ethernet), analyzes these benchmarks in the context of Lattice QCD, and outlines future plans for the project.

Only a rudimentary data transfer system between PC's has been implemented thus far. With this system, a set of consecutive integers is supposedly output from one PC to another, and the receiving PC's input buffer fills with data. However, this input buffer data bears no resemblance to the output data. Possible causes of this problem are proposed and analyzed.

Introduction

Motivations behind IP over IDE

Motivations for the present:

Certain QCD calculations on the lattice are very computationally intensive by today's standards. As a result, the community of Lattice QCD researchers are common users of the some of the world's fastest supercomputers. In the past year, Beowulf clusters have started to appear on the list of the world's fastest supercomputers (www.top500.org) and, due to their overall cost-effectiveness, will likely dominate the list in the future. Beowulf clusters that currently appear on this list use relatively expensive custom-designed network hardware, which take up a very significant fraction of the cluster's total cost. Thus the discovery of a more cost-effective form of high-performance networking for large Beowulf clusters could seriously improve the future adoption of such clusters as fast supercomputers, which would likely benefit the Lattice QCD community enormously.

Many Beowulf clusters* in existence today use gigabit-speed networking** to connect individual PC's together. The cost of hardware for this type of networking usually makes up a significant portion of the total cost of a cluster. For instance, the hardware necessary for the decentralized gigabit networking scheme used by Fodor *et al.* in 2002 made up about 30% of the total cost of their Beowulf cluster (1).

We find that for many applications (e.g., for certain QCD calculations on the lattice), decentralized gigabit networking appears to provide the most cost-effective option available today for new Beowulf clusters. In the networking scheme used by Fodor *et al.* (1) to perform Lattice QCD calculations, four gigabit Ethernet network cards are installed in each node (i.e., each PC in the cluster), which makes it possible to connect each node to four others. Figure 1 (next page) is a graphic representation of such a cluster.

---

† Internet Protocol – A data transfer protocol used to transfer data packets between devices on a network. This is the data transfer protocol of choice for the Internet.

‡ Integrated Disk Electronics – A technology that is used in most modern computers to transfer data between a computer's motherboard and a variety of internal data storage devices (such as hard disks, CD-ROM drives, DVD-ROM drives, and CD-RW drives).

Small Computer Systems Interface – A less common technology used in some modern computers to transfer data between a computer's motherboard and a variety of internal data storage devices (such as hard disks , CD-ROM drives, and CD-RW drives).

* A Beowulf cluster is a set of ordinary PC's (usually called nodes) connected over a high-speed network so that they may be used to perform parallel computations like a supercomputer.

** I.e., networking capable of attaining transfer rates of 1Gbit/sec, or about 128MB/sec.

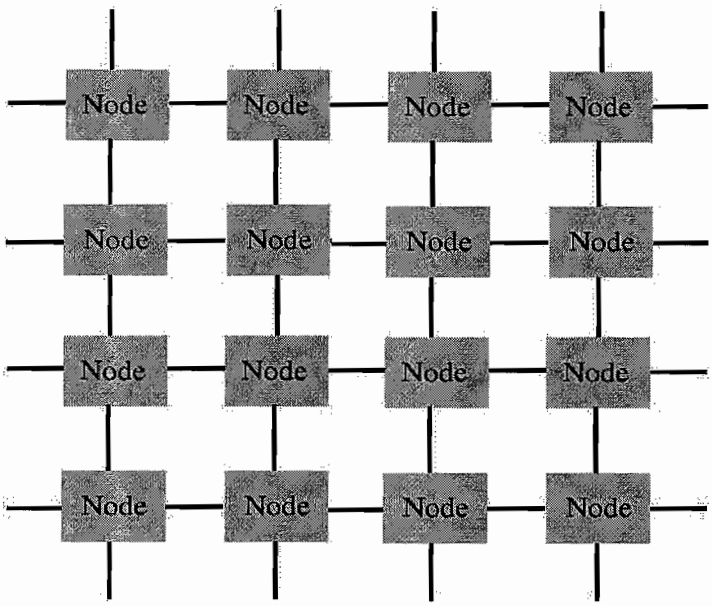


Figure 1: A decentralized networking scheme in which each node may connect to four others

The most apparent advantage of an IDE-based decentralized networking scheme over decentralized gigabit Ethernet networking schemes is its cost-effectiveness. In the above scenario (Figure 1), four gigabit cards are installed in each node. At the time of writing (April 2003), a gigabit card costs about $40, so the total cost per node for decentralized gigabit networking is about $160, neglecting the cost of cables.

In place of network cards, IP over IDE requires four IDE controllers per node in order for each node to connect to four others. Two IDE controllers are built-in to nearly all modern PC motherboards, so one only has to buy a single IDE controller card (which contains 2 IDE controllers) and four IDE cables. At the time of writing, a name-brand 133MB/sec IDE controller card costs about $20, and 36" round IDE cables cost $5 apiece. This brings the grand total to $40 per node for IP over IDE networking.

Therefore the IDE-based networking scheme just described would cost 25% as much as the decentralized gigabit Ethernet networking scheme per node. Further, if we are able to achieve a maximum data transfer rate close to the rated transfer rate on modern IDE controllers (133MB/sec) with average latency† on the order of Ethernet card latencies, IP over IDE would be possible with little to no loss in performance over decentralized gigabit networking.

Centralized gigabit Ethernet networking versus decentralized gigabit-speed networking:

In the simplest type of centralized gigabit Ethernet networking, every node is connected to a single gigabit Ethernet switch. Thus in order for a data packet to travel from one node to another, it only needs to travel through the switch. Every time a data packet travels from one network device to another (whether it be an Ethernet card, IDE controller, or Ethernet switch), it is called a hop. As the average number of hops necessary for a data packet to travel from one node to another increases, the average network latency increases, and the average network performance decreases. Thus centralized gigabit Ethernet networking is faster on average than

† In this paper, we define latency to be the amount of time necessary for a data packet to be transferred node to node, round-trip.

decentralized gigabit Ethernet networking.

The only catch to centralized networking is the cost, which increases roughly as the square of the number of ports. Currently (as of April 2003), a 20-port gigabit Ethernet switch costs about $2,500. We have stated above that a gigabit Ethernet card costs about $40, so we end up with a total cost of $165, neglecting the cost of cables. Surprisingly, this is only slightly more expensive per node than decentralized gigabit networking. However, one should keep in mind that gigabit switches with more ports would be significantly more expensive, since the cost per node for a single-switch centralized networking setup does not scale linearly with the number of nodes.

Motivations for the future

Conventional IDE as it exists today will more than likely be phased out in favor of Serial ATA within the next couple of years. Thanks to its design, Serial ATA will be able to transfer data much faster than current IDE technology. The fastest conventional IDE controller available today is able to transfer data at a peak rate of 133MB/sec. Serial ATA cards are currently available with peak data transfer rates of 150MB/sec, and according to Hachman (5), Serial ATA cards will be available with peak transfer rates of 300MB/sec by the beginning of 2005 and 600MB/sec by mid-2007.

IP over IDE Project History

In October of 2002, the author of this paper first proposed to Prof. Steven Gottlieb the possibility of using IDE cables to network a Beowulf cluster together in a decentralized fashion. Shortly after proposing the idea, an extensive Internet search was performed to see if such an idea had ever been implemented or patented. No previous IDE networking implementations or related patents were found, but a website devoted to an apparently successful Linux-based implementation of IP over SCSI (Small Computer Systems Interface) was found (2). It was decided that analyzing the software behind and setting up IP over SCSI would be an instructive first step, so the IP over SCSI software was downloaded and two SCSI cards were purchased. Within a day of receiving the cards, IP over SCSI was successfully working between two test nodes. Analysis and benchmarks of this networking scheme may be found below in the IP over SCSI section.

After the performance and design of IP over SCSI was briefly analyzed, the two nodes used in the IP over SCSI tests were connected over IDE and powered on. The first time this was tried, both nodes "froze up" when the BIOS (Basic I/O System, or the first piece of software a system runs after it is powered on) scanned for connected IDE hard disks. This was disconcerting, but without explanation, they booted just fine the second try. A detailed description of this problem with potential solutions are discussed in the "Current Status" section below. This problem was not considered to be serious enough to stop work on IP over IDE, so once the nodes were found to boot successfully, work on the software behind IP over IDE began.

It was known that the core of IP over IDE software would need to exist in kernel space (the software layer at the level of hardware drivers) and use original or slightly modified IDE I/O functions, so use of an operating system which allowed full access to IDE driver source code was preferable. The Linux operating system was chosen because it fits the above criterion and is the operating system of choice for most Beowulf clusters of which the author is aware. Thus the first step in the IP over IDE software project was to learn how to write and build a Linux kernel module. After a simple "Hello, kernel space!" module (the kernel module analogue of a "Hello, World!" program) was written and successfully compiled, analysis of the sparsely-commented IDE driver source code (in Linux kernel version 2.4.20) began. Here it must be pointed out that

this piece of source code was by far the most complex ever analyzed by the author of this paper. It took months to write modules capable of calling IDE driver I/O functions without causing segmentation faults or kernel crashes when loaded. Even though this point has now been reached, the goal of successfully sending and receiving a simple set of data over the IDE cable has not yet been attained. Details behind the current state of affairs in the IP over IDE project may be found below in the "IP over IDE: Current Status and Outstanding Problems" section. Future plans for this project are discussed in the below "IP over IDE: Future Goals" section.

IP over SCSI: Overview, Benchmarks, and Connection to IP over IDE

Overview

IP over SCSI software (2) sets up a network interface which enables one to send and receive data over SCSI cables using the Internet Protocol. With this setup, nearly all common networking software (e.g., ping, traceroute, ftp, etc.) may be used without modification to send and receive data through a SCSI card as though it were an Ethernet card.

In the context of the IP over IDE project, general similarities between IDE and SCSI do exist, so once we have reached the point at which data may be successfully transferred to/from nodes using IDE, the IP over SCSI software will likely provide a useful template for implementing the Internet Protocol over our IDE<->IDE connection. In its current form, IP over SCSI software consists of a Linux kernel patch for low-level SCSI driver modifications and a set of Linux kernel modules which access the low-level SCSI drivers and create a viable IP network interface for userspace programs (such as ping, traceroute, ftp, etc.)

Since a SCSI controller cannot send and receive data at the same time, a software-based token system must be used. Note that this is the same type of token as in "token ring". In its present form, IP over SCSI implements the token via a modification to the sym53c8xx SCSI device driver in the Linux kernel. This modification uses a kernel timer to check every 10ms whether the SCSI bus has been activated, so that both nodes do not attempt to activate the SCSI data pipeline at once. Based on this, we conclude that data packet should require at least 10ms at least to complete a round-trip. Since good documentation on IP over SCSI does not exist, here is an outline of what is believed to happen when the *ping* command is used on Node1 to send a 64byte data packet to Node2 and back:

1. Before the *ping* command is issued, each node checks every 10ms to see if the SCSI bus has been activated.
2. In response to the *ping* command, Node1 activates the SCSI bus with a write request. Node2 finds out that the SCSI bus has been activated and receives the data. The *ping* and the token timers start when the data has been sent.
3. Once it has received the data, Node2 immediately activates the SCSI bus with a write request.
4. 10ms after it first sent the packet, Node1 finds that the SCSI bus has been activated again, so it receives the data from Node2.
5. Node1 receives data while Node2 sends it. Node1 records approximately a 10ms round-trip time.

Using the *ping* command, we observed that ~98% of packets took 10.1ms to make a round trip, but some required about 20ms to complete the node to node circuit. Apparently, some packets required 2 timer ticks to complete a round trip. Until a more thorough analysis of the IP over SCSI software may be made, the exact cause of this anomaly will remain unknown.

Benchmarks

Test setup:

2 Pentium II 350MHz computers with identical hardware configurations were used in the benchmarks. Each node contains the following networking hardware:

- 1 LSI Logic ULTRA 160 SCSI Host Adapter PCI card (160MB/sec peak transfer rate)
- 1 Intel EtherExpress Pro 100Mbit Ethernet network card

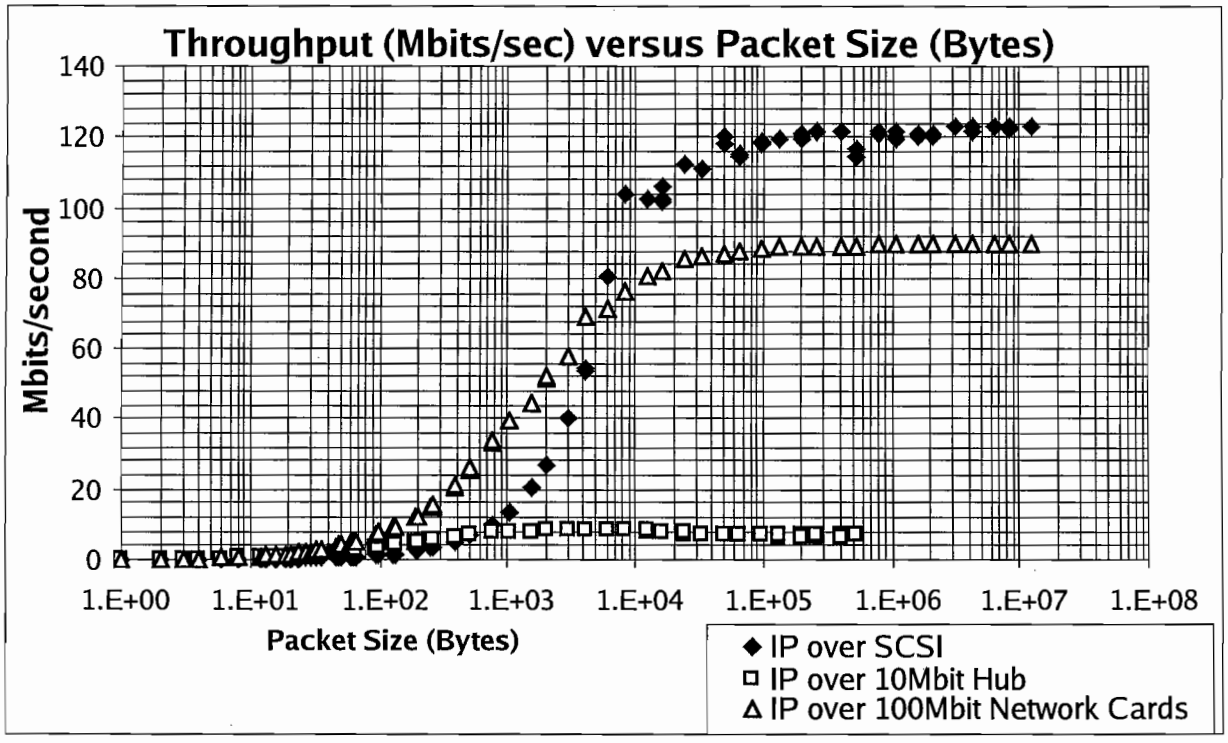
The nodes were connected with either a 10Mbit four-port Netgear Ethernet hub, a crossover cable, or a SCSI cable in the benchmarks.

Test methodology:

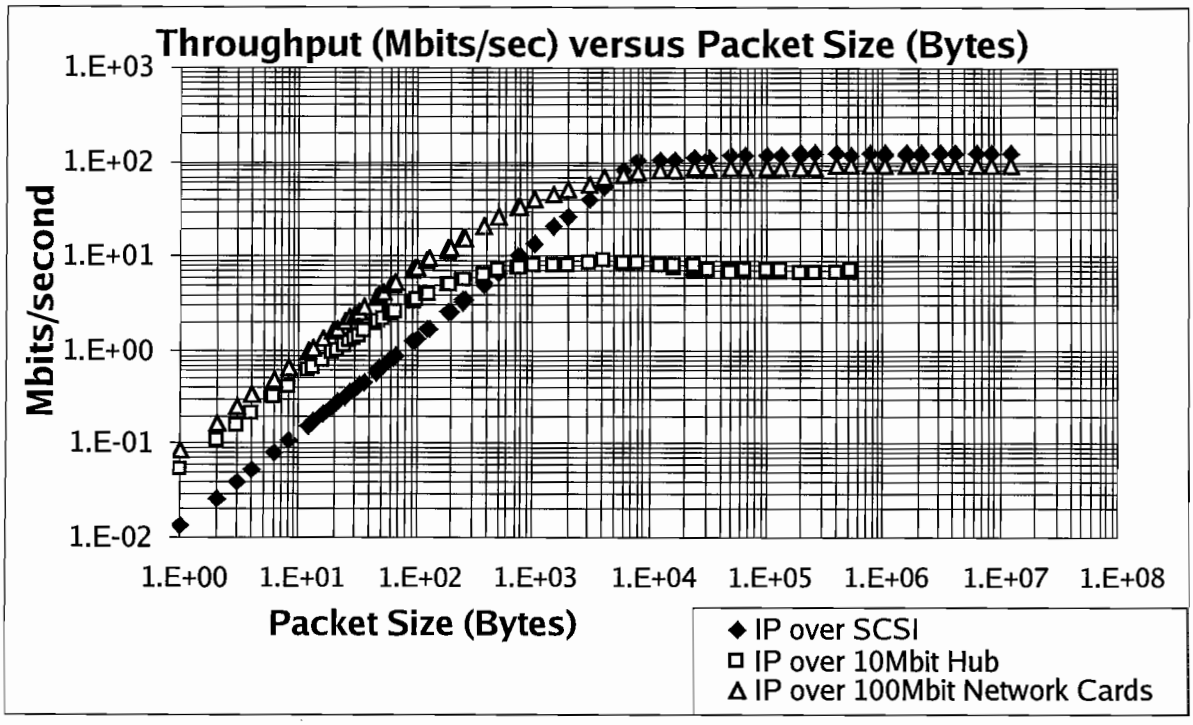
1. The nodes were first connected in one of the following 3 ways: over the hub, the crossover cable, or the SCSI cable.
2. The network modules corresponding to the desired connection method were loaded into the kernel and the corresponding network interface software was initialized.
3. NetPerf version 2.4 (3) was used to measure the data throughput from one node to the other as a function of packet size.
4. The nodes were either connected in a different way and the process was repeated starting with step 2, or if the nodes had already been connected in all 3 ways, the benchmarking process ended.

Results:

Note that NetPerf's computed error at each point in the below graphs yields error bars which are much smaller than any given point.



Graph 1: Semilog plot of throughput versus packet size. A linear scale is set on the vertical axis so that the maximum transfer rates for each networking scheme are easier to read.



Graph 2: Log-log plot of throughput versus packet size, using the same data as in Graph 1. Notice the linear increase in performance versus packet size at small packet sizes for all 3 networking schemes. This indicates that throughput increases exponentially with exponentially increasing packet size.

According to Graph 1, the maximum throughput allowed by IP over 10Mbit hub, IP over 100Mbit with crossover, and IP over SCSI was ~8Mbits/sec, ~90Mbits/sec, and ~122Mbits/sec, respectively. Notice that throughput seems to increase exponentially with exponentially increasing packet size up to ~8KB with IP over SCSI and 100Mbit over Crossover in Graph 1. To check whether this is the case, we input the data in Graph 1 into a log-log plot. The result is Graph 2.

According to Graph 2, 100Mbit networking provides the fastest networking up to the packet size of ~40KB. Below the packet size of 400Bytes, IP over SCSI provides the least throughput. We attribute this to the relatively large latency of the IP over SCSI software (10.1ms round trip ping time as compared to 0.29ms ping time over the hub). As the packet size increases, the amount of time spent transferring the data increases relative to the latency, so latency affects networking performance less at larger packet sizes.

IP over SCSI Performance Analysis in the Context of Lattice QCD

According to Prof. Steven Gottlieb, packet sizes of around 100B to 1KB are most common in LQCD algorithms, so the current technique used by IP over SCSI to transfer data packets provides no performance improvement over 100Mbit fast Ethernet networking for such algorithms. To avoid these large latency performance issues, IP over IDE software will need to implement either a modified token technique or an entirely new technique for sending and receiving packets.

IP over IDE: Current Status and Outstanding Problems

Current Status: General Procedure and Observations

First, the 2 nodes' cases are taken off and an IDE cable is used to connect the nodes' secondary IDE controllers together. The slave (middle) connector in the (3-connector) IDE cable is not used. For convenient access to both nodes at once, the nodes are connected over a 10Mbit hub to a controller computer, and an SSH (secure shell) daemon is run on all machines upon startup.

Next, Node1 is powered on, followed by Node2. For some reason, if Node2 is powered on first, the BIOS† on each node freezes up when the IDE bus is scanned for new hardware. When the nodes are connected over IDE, each node's BIOS discovers no hardware connected to the secondary IDE controller.

Much like the BIOS, the Linux kernel detects and configures hardware as it starts up. When it scans the IDE subsystem, the kernel does not detect any hardware connected to the secondary IDE controller. As a result, internal kernel data structures for this controller are not properly set up. Without setting up these kernel data structures properly, it may be impossible to transfer data between nodes over IDE using the current data transfer method (see discussion in the next two sections).

Once the nodes have successfully booted into Linux, we use the controller computer to connect to each node via SSH. Next, the input module is loaded on one of the nodes. This serves as a control. The input module reads 512B of integers from the IDE bus and outputs what it reads into the kernel log. Without fail, we observe that the input module reads 512B of zeros. This does not come as a surprise to us, since we believe that this indicates that the IDE bus is not active.

Next, the output module is loaded on one node, and immediately after that (about 0.5s later), the input module is loaded on the other node. The output module is supposed to send 512B worth of consecutive integers over the IDE cable, and once it has loaded, the input module no longer reads zeros. Instead of zeros or the consecutive integers, however, the input module reads 512 bytes of undecipherable gibberish. Probable causes behind and solutions to this problem are analyzed in the "Outstanding Problems" section below.

Current Status: Analysis of Functions used for IDE I/O in the Linux Kernel

Currently, IP over IDE software calls the Linux (version 2.4.20) kernel functions ide_input_data() and ide_output_data() to input and output data through the IDE bus. These functions are described in the kernel source code to perform PIO (programmed I/O) data transfers to/from the IDE interface. In PIO data transfers over IDE, data must pass through the CPU (central processing unit) on its way to/from the IDE bus, so this method requires a certain amount of CPU time while the transfer progresses. This is not desirable, since functions exist which transfer data at a faster rate to/from the IDE bus. These transfers also pass data through the DMA (direct memory access) controller chip off of the CPU, which allows for data to bypass the CPU altogether and not waste CPU time during data transfers.

Unfortunately, DMA I/O functions over IDE appear to require that certain data structures be properly set up when Linux is booted. As was discussed in the previous section, the Linux kernel does not bother to set certain variables for IDE interfaces when it does not detect hardware on the other end of the IDE cable. This poses many problems for the IP over IDE

† BIOS (basic input/output system) is the hardware detection and configuration software that a system runs after it is powered on.

project, and may be the cause of our current problem (*i.e.*, the input module reads indecipherable gibberish instead of the data that the other node sent across the cable).

Once we are able to successfully output data from one node and verify that the same data was received on the other node using PIO transfer functions, we plan to modify our kernel modules to properly initialize and use DMA data I/O functions.

Outstanding Problems: Analysis and Potential Solutions

BIOS freeze-up:

As stated before, both nodes boot properly when connected over IDE if and only if Node1 is powered on before Node2. Otherwise, the BIOS on each node freezes when the IDE bus is scanned for new hardware. Based on these observations, we predict that this problem would make it either very difficult or impossible to boot a decentralized network as shown in Figure 1.

To overcome this problem, we may need to install a modified version of LinuxBIOS (4) on all of the nodes. Currently used in many Beowulf clusters today, this program is a fast, free, and open source replacement BIOS for many modern PC's. The open source nature of this software allows it to be modified according to one's desires. If it is found that modifications to LinuxBIOS are necessary to fix our BIOS freeze-up problem, we would first need to modify the IDE detection algorithm to scan for and keep track of which IDE controllers on the local node are connected to IDE controllers on external nodes. For each of these IDE controllers with external connections, the hardware detection algorithm is skipped, thus fixing our BIOS freeze-up problem at IDE detection.

PIO data transfer difficulties:

When it is loaded, our IDE data output module (Appendix B) first initializes a 512B array of consecutive integers and then attempts to send it to the other node through the secondary IDE controller using the ide_output_data() function.

About 0.5s after the output module is loaded on one node, we load our IDE data input module (Appendix A) on the other node. When this module is loaded, it first initializes its own 512B input buffer, which is essentially an empty 512B array of integers. Next, the module uses the ide_input_data() function to fill its input buffer with data from the secondary IDE controller (which is connected to the other node over an IDE cable). Finally, this module outputs the contents of its "filled" input buffer to the kernel log file.

As described in the above section "General Procedure and Observations", when the above process takes place, the input module does not fill its input buffer with consecutive integers. Instead, the input buffer is filled with indecipherable gibberish. Note that this is different from the case when the output module is not loaded immediately before the input module. In that case, the input module fills its buffer with 512B of zeros.

After careful analysis, we believe that this problem is caused by the low-level output function inside ide_output_data() used to output data in our case. There are several low-level output functions are used by ide_output_data() to send data to a specified device on the IDE bus. ide_output_data() is written in such a way that only the optimal method of outputting data is used. To find this optimal method, ide_output_data() reads the values of certain variables in the primary IDE kernel data structure which indicate how quickly data may be sent to the specified device. In the case when none of these values are set, the slowest and perhaps buggiest low-level output function is used: outsw().

We know that variables in the main IDE data structure are set when the Linux kernel

scans the IDE controllers for connected devices at startup. We also know that the kernel does not set values for many of these variables if no device is detected. Since the kernel is unable to recognize a connection from the local IDE controller to an external one, it does not detect any devices on our controller at startup. Therefore, many important variables in the main IDE kernel data structure are not set for the controller we use to send data from one node to another, so ide_output_data() calls outsw() to send data from one node to another in our setup. In a way similar to that of ide_output_data(), ide_input_data() chooses the optimal input function from a set of low-level input functions. It calls the input function corresponding to the output function outsw() (insw()) to read from the specified IDE controller in our setup.

To fix this problem, we will likely need to assign values to improperly initialized variables in the main IDE kernel data structure. This will allow us to access other functions inside ide_output_data() and check whether we were correct in our assumption that outsw() and/or insw() are incapable of properly sending/receiving data through the IDE bus.

IP over IDE: Benchmarks

Test Methodology:

1. First, the header file <linux/delay.h> was added to to the IDE_out.c module from Appendix B. Functions from this file named do_gettimeofday() and mdelay() were also added to the IDE_out.c file. The former function returns the time of day in microseconds, and the latter function inserts a delay in milliseconds into the module. Although we only used one node in these benchmarks, we kept both nodes powered on and connected over IDE.
2. To determine how much time elapses between two successive calls of do_gettimeofday(), we called this function twice in a row and subtracted the first returned time from the second.
3. With a simple loop, we repeated the above measurement 100 times with a 10ms delay between each measurement.
4. Next, we measured the amount of time it takes to call ide_output_data() with a 512B output buffer by placing this function between two do_gettimeofday() function calls and subtracting the first returned time from the second.
5. With a simple loop, we repeated the above measurement 100 times with a 10ms delay between each measurement.
6. Assuming only random errors exist (there was no evidence to the contrary), the actual amount of time spent in the *i*'th ide_output_data() function call is equal to the *i*'th measured time in Step 4 minus the *i*'th measured time from Step 2.
7. Next, we compute the average of all 100 results from Step 6 and find the standard deviation of the mean.
8. Then we repeat Steps 4-7 with a 512B input or output buffer, calling the functions outsw(), ide_input_data(), and insw() instead.
9. Finally, we measure the amount of time necessary for a 512B packet to make a round-trip through a crossover cable between a 100Mbit Ethernet card in Node1 and another (of the same make and model) in Node2. To accomplish this, we execute the following command on Node2: *ping -s 512 -c 250 Node1*. This command will take 250 measurements. When the command has finished execution, it outputs the average time in milliseconds and the standard deviation of a single measurement.

Results:

The average time computed in Step 2 (above) over 100 measurements was (0.31+/-0.05μs). We then followed the rest of the procedure outlined above and entered our results into the below table.

Type of networking	Function or command used	Number of Measurements	Mean time per measurement (μs)	Standard deviation of the mean (μs)
IDE, PIO transfers	ide_output_data()	100	231.5	0.3
IDE, PIO transfers	outsw()	100	231.9	0.3
IDE, PIO transfers	ide_input_data()	100	227.9	0.3
IDE, PIO transfers	insw()	100	227.9	0.3
IP over 100Mbit Ethernet cards connected over crossover cable	ping -c 250 -s 512 Node1 (executed on Node2)	250	198	1

Table 1: I/O timings versus function call and networking type.

Comparisons and analysis:

Notice in Table 1 that on average, it takes about 4μs longer to output a 512B buffer than it does to fill an input buffer of the same size using IDE I/O functions. This result is unexpected and might have been sufficient cause for investigation if we wanted to continue using insw() and outsw() in the future.

The fact that ide_output_data() measurement times are approximately the same as outsw() measurement times on average comes as no surprise, since we concluded in the above "Outstanding Problems" section that ide_output_data() calls outsw() anyway. Likewise, for ide_input_data() and insw(), we get approximately the same measurements since ide_input_data() calls insw() (as concluded in the above "Outstanding Problems" section).

We know that a round-trip packet through IDE would require an output from Node1, followed by an input from Node2, followed by an output from Node2, followed by an input from Node1. Based on the data above and what was just stated, we conclude that sending a packet round-trip through IDE with the I/O functions used in Table 1 would take at least 918ms. Since the *ping* command sends packets round-trip, we conclude that if IP over IDE were to use these functions and a 512B packet size, it would be about 5 times slower on average than 100Mbit networking over a crossover cable with the same packet size. This is totally unacceptable, since many parallel Lattice QCD algorithms use packet sizes on the order of 512B (according to Prof. Steven Gottlieb).

It should be pointed out that there exists one large caveat in interpreting the above results from the IDE I/O functions. Since we have yet to decipher the data output by the output module using the input module on another node, we don't know if 512B of data are actually being output or input through the IDE controller. The large function call times may in fact be due to some hardware send or receive data timeout. We hope that use of different IDE I/O functions will allow us to successfully transfer data between nodes and accomplish this with lower function call times.

IP over IDE: Future Goals

Fixing Current Problems

As stated before, we believe that the lack of configuration data in the IDE kernel configuration data structure is a primary cause of our current problems. Without proper configuration data for hdc, our kernel modules (designed for Linux version 2.4.20) are unable to send data through the IDE bus without using the (possibly buggy) outsw() and insw() functions. We have recently learned that the IDE subsystem in the 2.5.x beta series of the Linux kernel has been revamped to a large extent and now includes functions which allow modules to add and remove settings in the main IDE configuration data structure at will. With this new functionality, we conclude that the 2.5.x beta series of the Linux kernel may provide us with the ability to properly set up the IDE configuration data necessary to call certain IDE I/O functions. As far as IDE I/O functions in the 2.5.x beta series are concerned, ide_input_data() and ide_output_data() no longer exist. They have been superseded by a single IDE I/O function called generic_ide_ioctl().

In an attempt to overcome current problems with the kernel modules in Appendices A and B, we plan to rewrite them to support Linux 2.5.x. This will not be an easy task. Around the beginning of April 2003, compilation of a simple "Hello, kernel space!" kernel module was tried with Linux 2.5.66. Despite repeated attempts, this simple module would not compile. This is the first problem we must solve before progress in writing new kernel modules will be made.

Once the compilation problem has been fixed for a simple "Hello, kernel space!" kernel module, we will need to find out how to call the generic_ide_ioctl() function to input and output data through the cable connected to the other node. Once the input and output modules are able to call this I/O function and load without error, we will try to send some simple data from one node to another. If we run into the same problems as before, we will probably need to use the new functions in 2.5.x that enable us to access and modify the IDE configuration data structure.

Serial ATA

We recently purchased two Serial ATA cards with a rated peak transfer rate of 150MB/sec. Each Serial ATA card contains two Serial ATA controllers. These controllers are supported by the newest 2.5.x beta versions of the Linux kernel. In such versions of the Linux kernel, Serial ATA drivers use the same kernel IDE configuration data structure and the same generic IDE I/O functions as conventional IDE. Thus with little to no modifications, we should be able to convert our Serial ATA data I/O kernel modules into conventional IDE data I/O kernel modules.

Once we are able to successfully output and input a single data packet between nodes using Serial ATA, we plan to first conduct a comprehensive set of latency benchmarks and compare these results with the results in Table 1. We will then modify the I/O modules to support conventional IDE (if modifications are necessary), repeat the same set of latency benchmarks, and compare with Table 1 and Serial ATA benchmark results. At this point, an IP over IDE web page will be created and the Open Source community will be asked to join in the IP over IDE effort. The source code, which is licensed under the GNU General Public License, will be available for download on this website. We believe that successfully transferring a data packet between nodes constitutes the "proof of concept" necessary for the information about the IP over IDE project to spread and successfully recruit new developers.

Finally, with the help of the Open Source community, we plan to implement IP over IDE, possibly using IP over SCSI source code as a template. We will likely perform many benchmarks along the way to determine whether decentralized IP over IDE networking could

become a viable alternative to decentralized gigabit Ethernet networking.

The author of this paper would like to thank Prof. Steven Gottlieb for his unfailing support and invaluable help in this project. This research was supported by grants from the Indiana University Honors College and the Indiana University Office of Research and University Graduate School.

References

(1): Fodor, Z., *et al.* "Better than $1/Mflops sustained: a scalable PC-based parallel computer for lattice QCD". http://xxx.lanl.gov/abs/hep-lat/0202030.
(2): Semeano, Pedro *et al.* http://ipoverscsi.sourceforge.net/.
(3): Jones, Rick *et al.* http://www.netperf.org/.
(4): Minnich, Ron *et al.* http://linuxbios.org.
(5): Hachman, Mark. "Serial ATA Could Be Slow To Take Off". http://www.extremetech.com/article2/0,3973,907055,00.asp.

Appendix A: IDE_in.c

```
/*
 *  IDE data input algorithm Version 0.0.2, April 17, 2003
 *  By Zach Etienne, with help and support from Prof. Steven Gottlieb
 *  This software is licensed under the GNU General Public License (GPL).
 *  The GPL may be found at http://www.gnu.org/licenses/licenses.html.
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/ide.h>

#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif
#define NR_IRQS 16

/*
 * ide_hwifs[] is the main IDE configuration data structure
 */
extern ide_hwif_t ide_hwifs[];

//The following function is initialized when this module is loaded:
int init_module(void) {
  int index, drindex, dummy_i;
  unsigned int packetsize=512;  //packet size in Bytes

  ide_hwif_t *hwif;  //With this pointer, we later choose ide1
                     //(secondary IDE controller)
  ide_drive_t *drive; //With this pointer, we later choose hdc (master)

  int h[packetsize/sizeof(int)]; //Initialize input buffer.

  index=1; //Later Chooses ide1, which is the secondary IDE controller
  drindex=0; //Later Chooses hdc, which is the secondary master "drive"

  hwif = &ide_hwifs[index];    //Here is where we choose ide1
  drive = &hwif->drives[drindex];  //Here is where we choose hdc

  //The following printk's output debugging information about
  //which drive and controller we have selected.
```