JC377 U.S. PTO
08/27/98

A/Prov

JC541 U.S. PTO
60/098296
08/27/98

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Application of Laurence B. Boucher et al. Docket No: ALA-004

Filing Date: August 27, 1998 Express Mail Label No: EE237180694US

Title: INTELLIGENT NETWORK INTERFACE DEVICE
AND SYSTEM FOR ACCELERATED COMMUNICATION

August 27, 1998

Box Provisional Application
Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

This is a request for filing the above-referenced, attached PROVISIONAL APPLICATION FOR PATENT under CFR 1.53(b)(2). The inventors are listed below:

Inventors	Residence
Laurence B. Boucher	Saratoga, California
Stephen E. J. Blightman	San Jose, California
Peter K. Craft	San Francisco, California
David A. Higgin	Saratoga, California
Clive M. Philbrick	San Jose, California
Daryl D. Starr	Milpitas, California

60098296.082798

The application contains a 173 page Specification which includes interspersed Drawings.

Please address all correspondence to the address below.

Enclosed please find a check in the amount of $150.00 to cover the Filing Fee.

Respectfully submitted,

[illegible]
Mark Lauer
Reg. No. 36,578
6850 Regional Street
Suite 250
Dublin, CA 94568
Tel: (925) 556-3500
Fax: (925) 803-8189

CERTIFICATE OF MAILING

I hereby certify that this correspondence is being deposited with the United States Postal Service, *Express Mail Post Office to Addressee*, Label No. EE237180694US, addressed to: Box Provisional Application, Assistant Commissioner for Patents, Washington, D.C. 20231, on August 27, 1998.

Date: 8-27-98

[illegible]
Mark Lauer

INTELLIGENT NETWORK INTERFACE DEVICE AND SYSTEM FOR ACCELERATED COMMUNICATION

Provisional Patent Application Filed Under 35 U.S.C. § 111 (b)

Inventors: Laurence B. Boucher
Stephen E. J. Blightman
Peter K. Craft
David A. Higgin
Clive M. Philbrick
Daryl D. Starr

Assignee: Alacritech Corporation

1. Background of the Invention

Network processing as it exists today is a costly and inefficient use of system resources. A 200 MHz Pentium-Pro is typically consumed simply processing network data from a 100Mb/second-network connection. The reasons that this processing is so costly are described in the next few pages.

1.1. Too Many Data Moves

When network packet arrives at a typical network interface card (NIC), the NIC moves the data into pre-allocated network buffers in system main memory. From there the data is read into the CPU cache so that it can be checksummed (assuming of course that the protocol in use requires checksums. Some, like IPX, do not.). Once the data has been fully processed by the protocol stack, it can then be moved into its final destination in memory. Since the CPU is moving the data, and must read the destination cache line in before it can fill it and write it back out, this involves at a minimum 2 more trips across the system memory bus. In short, the best one can hope for is that the data will get moved across the system memory bus 4 times before it arrives in its final destination. It can, and does, get worse. If the data happens to get invalidated from system cache after it has been checksummed, then it must get pulled back across the memory bus before it can be moved to its final destination. Finally, on some systems, including Windows NT 4.0, the data gets copied yet another time while being moved up the protocol stack. In NT 4.0, this occurs between the miniport driver interface and the protocol driver interface. This can add up to a whopping 8 trips across the system memory bus (the 4 trips described above, plus the move to replenish the cache, plus 3 more to copy from the miniport to the protocol driver). That's enough to bring even today's advanced memory busses to their knees.

1.2. Too Much Processing by the CPU

In all but the original move from the NIC to system memory, the system CPU is responsible for moving the data. This is particularly expensive because while the CPU is moving this data it can do nothing else. While moving the data the CPU is typically stalled waiting for the relatively slow memory to satisfy its read and write requests. A CPU, which can execute an instruction every 5 nanoseconds, must now wait as long as several hundred nanoseconds for the memory controller to respond before it can begin its next instruction. Even today's advanced pipelining technology doesn't help in these situations because that relies on the CPU being able to do useful work while it waits for the memory controller to respond. If the only thing the CPU has to look forward to for the next several hundred instructions is more data moves, then the CPU ultimately gets reduced to the speed of the memory controller.

Moving all this data with the CPU slows the system down even after the data has been moved. Since both the source and destination cache lines must be pulled into the CPU cache when the data is moved, more than 3k of instructions and or data resident in the CPU cache must be flushed or invalidated for every 1500 byte frame. This is of course assuming a combined instruction and data second level cache, as is the case with the Pentium processors. After the data has been moved, the former resident of the cache will likely need to be pulled back in, stalling the CPU even when we are not performing network processing. Ideally a system would never have to bring network frames into the CPU cache, instead reserving that precious commodity for instructions and data that are referenced repeatedly and frequently.

But the data movement is not the only drain on the CPU. There is also a fair amount of processing that must be done by the protocol stack software. The most obvious expense is calculating the checksum for each TCP segment (or UDP datagram). Beyond this, however, there is other processing to be done as well. The TCP connection object must be located when a given TCP segment arrives, IP header checksums must be calculated, there are buffer and memory management issues, and finally there is also the significant expense of interrupt processing which we will discuss in the following section.

1.3. Too Many Interrupts

A 64k SMB request (write or read-reply) is typically made up of 44 TCP segments when running over Ethernet (1500 byte MTU). Each of these segments may result in an interrupt to the CPU. Furthermore, since TCP must acknowledge all of this incoming data, it's possible to get another 44 transmit-complete interrupts as a result of sending out the TCP acknowledgements. While this is possible, it is not terribly likely. Delayed ACK timers allow us to acknowledge more than one segment at a time. And delays in interrupt processing may mean that we are able to process more than one incoming network frame per interrupt. Nevertheless, even if we assume 4 incoming frames per input, and an acknowledgement for every 2 segments (as is typical per the ACK-every-other-segment property of TCP), we are still left with 33 interrupts per 64k SMB request.

Interrupts tend to be very costly to the system. Often when a system is interrupted, important information must be flushed or invalidated from the system cache so that the interrupt routine instructions, and needed data can be pulled into the cache. Since the CPU will return to its prior location after the interrupt, it is likely that the information flushed from the cache will immediately need to be pulled back into the cache.

What's more, interrupts force a pipeline flush in today's advanced processors. While the processor pipeline is an extremely efficient way of improving CPU performance, it can be expensive to get going after it has been flushed.

Finally, each of these interrupts results in expensive register accesses across the peripheral bus (PCI). This is discussed more in the following section.

1.4. Inefficient Use of the Peripheral Bus (PCI)

We noted earlier that when the CPU has to access system memory, it may be stalled for several hundred nanoseconds. When it has to read from PCI, it may be stalled for many microseconds. This happens every time the CPU takes an interrupt from a standard NIC. The first thing the CPU must do when it receives one of these interrupts is to read the NIC Interrupt Status Register (ISR) from PCI to determine the cause of the interrupt. The most troubling thing about this is that since interrupt lines are shared on PC-based systems, we may have to perform this expensive PCI read even when the interrupt is not meant for us!

There are other peripheral bus inefficiencies as well. Typical NICs operate using descriptor rings. When a frame arrives, the NIC reads a receive descriptor from system memory to determine where to place the data. Once the data has been moved to main memory, the descriptor is then written back out to system memory with status about the received frame. Transmit operates in a similar fashion. The CPU must notify that NIC that it has a new transmit. The NIC will read the descriptor to locate the data, read the data itself, and then write the descriptor back with status about the send. Typically on transmits the NIC will then read the next expected descriptor to see if any more data needs to be sent. In short, each receive or transmit frame results in 3 or 4 separate PCI reads or writes (not counting the status register read).

2. Summary of the Invention

Alacritech was formed with the idea that the network processing described above could be offloaded onto a cost-effective Intelligent Network Interface Card (INIC). With the Alacritech INIC, we address each of the above problems, resulting in the following advancements:

The vast majority of the data is moved directly from the INIC into its final destination. A single trip across the system memory bus.

There is no header processing, little data copying, and no checksumming required by the CPU. Because of this, the data is never moved into the CPU cache, allowing the system to keep important instructions and data resident in the CPU cache.

Interrupts are reduced to as little as 4 interrupts per 64k SMB read and 2 per 64k SMB write.

There are no CPU reads over PCI and there are fewer PCI operations per receive or transmit transaction.

The remainder of this document will describe how we accomplish the above.

2.1. Perform Transport Level Processing on the INIC

In order to keep the system CPU from having to process the packet headers or checksum the packet, we must perform this task on the INIC. This is a daunting task. There are more than 20,000 lines of C code that make up the FreeBSD TCP/IP protocol stack. Clearly this is more code than could be efficiently handled by a competitively priced network card. Furthermore, as we've noted above, the TCP/IP protocol stack is complicated enough to consume a 200 MHz Pentium-Pro. In order to perform this function on an inexpensive card, we need special network processing hardware as opposed to simply using a general purpose CPU.

2.1.1. Focus On TCP/IP

In this section we introduce the notion of a "context". A context is required to keep track of information that spans many, possibly discontiguous, pieces of information. When processing TCP/IP data, there are actually two contexts that must be maintained. The first context is required to reassemble IP fragments. It holds information about the status of the IP reassembly as well as any checksum information being calculated across the IP datagram (UDP or TCP). This context is identified by the IP_ID of the datagram as well as the source and destination IP addresses. The second context is required to handle the sliding window protocol of TCP. It holds information about which segments have been sent or received, and which segments have been acknowledged, and is identified by the IP source and destination addresses and TCP source and destination ports.

If we were to choose to handle both contexts in hardware, we would have to potentially keep track of many pieces of information. One such example is a case in which a single 64k SMB write is broken down into 44 1500 byte TCP segments, which are in turn broken down into 131 576 byte IP fragments, all of which can come in any order (though the maximum window size is likely to restrict the number of outstanding segments considerably).

Fortunately, TCP performs a Maximum Segment Size negotiation at connection establishment time, which should prevent IP fragmentation in nearly all TCP connections. The only time that we should end up with fragmented TCP connections is when there is a router in the middle of a connection which must fragment the segments to support a smaller MTU. The only networks that use a smaller MTU than Ethernet are serial line interfaces such as SLIP and PPP. At the moment, the fastest of these connections only run at 128k (ISDN) so even if we had 256 of these connections, we would still only need to support 34Mb/sec, or a little over three 10bT connections worth of data. This is not enough to justify any performance enhancements that the INIC offers. If this becomes an issue at some point, we may decide to implement the MTU discovery algorithm, which should prevent TCP fragmentation on all connections (unless an ICMP redirect changes the connection route while the connection is established). With this in mind, it seems a worthy sacrifice to not attempt to handle fragmented TCP segments on the INIC.

SPX follows a similar framework as TCP, and so the expansion of the INIC to handle IPX/SPX messages is straightforward.

UDP is another matter. Since UDP does not support the notion of a Maximum Segment Size, it is the responsibility of IP to break down a UDP datagram into MTU sized packets. Thus, fragmented UDP datagrams are very common. The most common UDP application running today is NFSV2 over UDP. While this is also the most common version of NFS running today,

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.