

Introduction

Ethernet has become a standard data transport paradigm for embedded systems across all applications. The reason for selecting Ethernet is simple—the transport technology is (relatively) cheap, abundant (networks are everywhere), mature, and reliable.

The primary goal of this application note is to provide you with techniques to accelerate the performance of your Nios® II networking application, and to show how certain key optimizations can improve the overall performance of the system. The result of applying these optimizations is noted in the benchmarking results section of this application note. The secondary goal is to provide you with a better understanding of how the different parts of a Nios II Ethernet-enabled system work together, and how the interaction of these parts correspond to the total networking performance of the system.

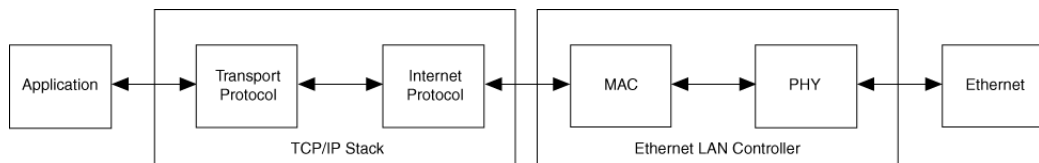
The Structure of Networking Applications

This section describes the different parts of a general networking application.

Ethernet System Hierarchy

Figure 1 shows the flow of information from an embedded networking application to the Ethernet.

Figure 1. The Ethernet System Hierarchy



The structure presented in Figure 1 shows a typical embedded networking system. In general, a user application performs a job that defines the goal of the embedded system (for example, controlling the speed of a motor, serving as the UI for an embedded kiosk, and so forth). The application uses an API (generally Sockets) provided by the networking stack to send networking data to and from the embedded system.

The stack itself is a software library that converts data from the user application into networking packets, and sends the packets via the networking device. Networking stacks tend to be very complicated software state machines that must be able to send data using a wide variety of networking protocols (ARP, TCP, UDP, and so forth). These stacks generally require a significant amount of processing power from the CPU to get their job done.

The Ethernet device is used by the stack to move data across the physical media. Most of a networking stack's interaction with the networking device consists of shuttling Ethernet packets to and from the Ethernet device.

The link layer, or physical media upon which the Ethernet datagrams traverse, cannot be ignored when constructing a network enabled system. Depending on the location of the embedded system, the Ethernet datagrams may be traversing over a wide variety of physical links (10/100 Mbit twisted pair, fiber optic, and so forth). Additionally, the datagrams may experience latency if traversing long distances or need to be broadcast through many network switches in order to arrive at their destination.

Inter-Relationship of Elements

The total throughput performance of an embedded networking system is highly dependent on the interaction of the user application, networking stack, Ethernet device (and driver), as well as the physical connection for the networking link. Making substantial performance improvements in the network throughput often depends on optimizing the performance of all these elements simultaneously.

In general, your networking application has some criteria for performance that are either achieved or not. However, a good first order approximation for determining the viability of your networking application is to remove the user application from the system and measure the total networking performance. This provides you with an "upper bound" of total network performance, which you can use to create your networking application. This application note uses a simple benchmark program that determines the "raw" throughput rate of TCP and UDP data transactions. This benchmark application does very little apart from sending or receiving data through the networking stack. It therefore provides us with a good approximation of the maximum networking performance achievable.

Finding the Performance Bottlenecks

There are a wide variety of tools available for analyzing the performance of your Nios II embedded system and finding system bottlenecks. In this application note, many of the techniques presented to increase overall system (and networking) performance were discovered through the use of these tools. While this application note doesn't explore the use of these tools, and how they were applied to find networking bottlenecks in the system, they are listed here for your information:

- GNU Profiler
- SOPC Builder Timer Peripheral
- SOPC Builder Performance Counters



For more information about finding general performance bottlenecks in your Nios II embedded system, refer to *AN 391: Profiling Nios II Systems*.

The User Application

In an embedded networking system, the application layer is the part of the system where your “key task” is being completed. In general, this application layer performs some work and then uses the network stack to send and receive data. In a classic embedded networking system, your application is being executed on the same CPU as the network stack, and is also vying for computation resources.

To increase the throughput of your networking system, decrease the time your application spends completing its task between the function calls it makes to the networking stack. There is a two-fold benefit to doing this. First, the faster your application runs to completion before sending or receiving data, the more function calls it can make to the networking stack (Sockets API) to move data across the network. Second, if the application takes less of the processor's time to run, the more time the processor has to operate the networking stack (and networking device) and transmit the data.

User Application Optimizations

This section describes some effective ways to decrease the amount of time your application uses the Nios II CPU.

Software Optimizations

- **Compiler Optimization Level**—Compile your application with the highest compiler optimization possible, -O3. Higher optimizations result in denser, more highly optimized code, thereby increasing the computational efficiency of the processor.

- **MicroC/OS-II Thread Priority**—Make sure that your application task has the right MicroC/OS-II priority level assigned to it. In general, the higher the priority of the application, the faster it runs to completion. The application's priority levels should be balanced against the priority levels assigned to the NicheStack's core tasks discussed in “[Structure of the NicheStack Networking Stack](#)” on page 8.



This suggestion assumes that your application is using Altera's recommended method for operating the NicheStack Networking Stack, which requires using the MicroC/OS-II operating system.

Hardware Optimizations

- **Processor Performance**—The performance of the Nios II processor can be increased in several ways:
 - **Computational Efficiency**—Selecting the most computationally efficient Nios II processor core is the quickest way to improve overall application performance. The available Nios II processor cores, in order of performance, are the Nios II/f core (fastest), the Nios II/s core (standard), and the Nios II/e core (slowest).
 - **Memory Bandwidth**—Using low-latency, high speed memory decreases the amount of time required by the processor to fetch instructions and move data. Additionally, increasing the processor's arbitration “share” of the memory via SOPC Builder increases the processor's performance by allowing the Nios II processor to perform more transactions to the memory before another Avalon master can assume control of the memory.
 - **Instruction/ Data Caches**—Adding an instruction and data cache is an effective way to decrease the amount of time the Nios II processor spends performing operations, especially in systems that have slow memories (DDR SDRAM, SDRAM, and so forth). In general, the larger the cache size selected for the Nios II processor, the greater the performance improvement.
 - **Clock Frequency**—Increasing the speed of the processor's clock results in more instructions being executed per unit of time. To gain the best performance possible, you should ensure that the processor's execution memory is on the same clock domain as the processor, to avoid the use of clock-crossing FIFOs.

One of the easiest ways to increase the operational clock frequency of the processor and memory peripherals is to use an SOPC Builder FIFO bridge peripheral to isolate the slower peripherals of the system. With this peripheral, the processor, memory, and Ethernet device are connected on one side of the

bridge. On the other side of the bridge are all of the peripherals that are not performance dependent. The optimized Ethernet design, which appears in the benchmark section of this document, uses a FIFO bridge for this reason.

- **Hardware Acceleration**—Hardware acceleration can provide tremendous performance gains by moving time-intensive processor tasks to dedicated hardware blocks in the system. The three most common ways to accelerate application level algorithms are as follows:
 - **Custom Instruction**—Off-loads the Nios II CPU by using hardware to implement a custom instruction.
 - **C2H (C to Hardware) Accelerator**—Accelerates processor execution by converting C algorithms into hardware subroutines.
 - **Custom Peripheral**—Create a block of hardware that performs a specific algorithmic task, controllable from the Nios II CPU, as a peripheral.

Because every application has different computational needs, this application note does not focus on these technologies in any great detail.

The Sockets API

After “tuning” your application to become more computationally efficient (thereby freeing more of the CPU’s time for operating the networking stack), the next area of optimization comes from how your application uses the networking stack. This section describes how to select the best protocol for use by your application and the most efficient way to use the Sockets API.

Selecting the “Right” Networking Protocol

When using the Sockets API, you must also select which protocol to use for transporting data across the network. There are two main protocols used to transport data across networks: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). Both of these protocols

perform the basic function of moving data across Ethernet networks, but they have very different implementations and performance implications. [Table 1](#) compares the two protocols.

Table 1. The UDP and TCP Protocols

Parameter	Protocol	
	UDP	TCP
Connection Mode	Connection-Less	Connection-Oriented
In Order Data Guarantee	No	Yes
Data Integrity and Validation	No	Yes
Data Retransmission	No	Yes
Data Checksum	Yes; Can be disabled	Yes

In terms of just throughput performance, the UDP protocol is much faster than TCP because it has very little overhead. The UDP protocol makes no attempt to validate that the data being sent arrived at its destination (or even that the destination is capable of receiving packets), so the network stack needs to perform much less work in order to send or receive data using this protocol.

However, aside from very specialized cases where your embedded system can tolerate losing data (for example, streaming multi-media applications), use the TCP protocol.



Design Tip: Use the UDP protocol to gain the fastest performance possible; however, use the TCP protocol when you must guarantee the transmission of the data.

Improving Send/Recv Performance

Proper use of the Sockets API in your application can also increase the overall networking throughput of your system. Following are several ways to optimally use the Sockets API:

- **Minimize Send/Recv function calls**—The Sockets API provides two sets of functions for sending and receiving data through the networking stack. For the UDP protocol these functions are `sendto` and `recvfrom`. In the TCP protocol these functions are `send` and `recv`.

Depending on which transport protocol you use (TCP or UDP), your application uses one of these sets of functions. To increase overall performance, avoid calling these functions repetitively to handle small units of data. Every call to these functions incurs a fixed time

penalty for execution, which can compound quickly when these functions are called multiple times in rapid succession. Instead, you should aggregate data you want to send (or receive) and call these functions with the largest possible amount of data at one time to send or receive.



Design Tip: Call the Socket's send and receive functions with larger buffer sizes to minimize system call overhead.

- **Minimize Latency in Sending Data**—Although the TCP Sockets send function can accept an arbitrary number of bytes, those bytes may not be immediately sent as a packet. This is especially true when send is called with a small number of bytes because the networking stack attempts to coalesce these small data “chunks” into a larger packet. This is done to avoid congesting the network with many small packets (using the Nagle Algorithm for congestion avoidance). There is a solution, however, through the use of the TCP_NO_DELAY flag.

When a socket has its TCP_NO_DELAY flag set via the `setsockopt` function call, the Nagle Algorithm is disabled and the socket immediately sends whatever bytes are passed in as a TCP packet. This can be a useful way to increase network throughput in the case where your application must send many small “chunks” of data very quickly.



Design Tip: If you need to accelerate the transmission of small TCP packets, use the TCP_NO_DELAY flag on your socket. An example of setting the TCP_NO_DELAY flag can be found in the benchmarking application software, found in the downloadable reference design.



While disabling the Nagle Algorithm should cause smaller packets to be immediately sent over the network, the networking stack may still coalesce some of the packets into larger packets. This is especially true in the case of the Windows workstation platform. The networking stack should, however, do so with much lower frequency than if the Nagle Algorithm was enabled.

The “Zero Copy” API

The NicheStack networking stack provides a further optimization to accelerate the data transfers to and from the stack called the “Zero Copy” API. The “Zero Copy” API increases overall system performance by eliminating the buffer management scheme performed by the Socket

API's read and write function calls. The send and receive data buffers are directly managed by the user's application, thereby eliminating an extra level of data copying performed by the Nios II CPU.

This performance optimization is not covered in detail in this application note. Refer to the ["Appendix" on page 25](#) for pointers to more information.



Design Tip: Using the NicheStack "Zero Copy" API may accelerate your network application's throughput by eliminating an extra layer of copying.

Structure of the NicheStack Networking Stack

The NicheStack networking stack is a highly configurable software library designed for communicating over TCP/IP networks. The version that Altera ships in the Nios II EDS has been optimized for use with the MicroC/OS-II Real Time Operating System (RTOS), and includes device driver support for the LAN91C111 and Altera® Triple Speed Ethernet (TSE) MAC.

The NicheStack networking stack is extremely configurable, with the entire software library utilizing a single configuration header file, called **ippport.h**.

General Optimizations

Because this application note focuses on a single Nios II system, most of the optimizations described in ["User Application Optimizations" on page 3](#) also improve the performance of the NicheStack networking stack. The following optimizations also help increase your overall network performance.

Software optimizations:

- Compiler Optimization Level

Hardware optimizations:

- Processor Performance
 - Computational Efficiency
 - Memory Bandwidth
 - Instruction/ Data Caches
 - Clock Frequency

NicheStack Specific Optimizations

There are several targeted optimizations that can be used to increase the performance of the NicheStack networking stack directly. These improvements are described below.

NicheStack Thread Priorities

Altera's version of the NicheStack networking stack relies on the MicroC/OS-II operating system's threads to drive two critical tasks to properly service the networking stack. These tasks (threads) are **tk_nettick**, which is responsible for time-keeping, and **tk_netmain**, which is used to drive the main operation of the stack.

When building a NicheStack-based system in the Nios II EDS, the default run-time thread priorities assigned to these tasks are: **tk_netmain** = 2 and **tk_nettick** = 3. These thread priorities have been selected to provide the best networking performance possible for your system. However, in your embedded system you may need to override these priorities because your application task (or tasks) run more frequently than these tasks. Doing this, however, may result in performance degradation of network operations, as the NicheStack networking stack has less processor cycles to complete its tasks.

Therefore, if you need to increase the priority of your application tasks above that of the NicheStack tasks, make sure to yield control whenever possible to ensure that these tasks get some processor time. Additionally, ensure that the **tk_netmain** and **tk_nettick** tasks have priority levels that are just slightly less than the priority level of your critical system tasks.

When you yield control, your application task is placed from a “running” state into a “waiting” state by the MicroC/OS-II scheduler, which then takes the next “ready” task and places it into a running “state.” If **tk_netmain** and **tk_nettick** are the higher priority tasks, they are allowed to run more frequently, which in turn increases the overall performance of the networking stack.



Design Tip: If your MicroC/OS-II based application tasks run with a higher priority level (lower priority number) than the NicheStack tasks, remember to yield control periodically so the NicheStack tasks can run. Tasks using the NicheStack services should call the function **tk_yield**. If they are not using the NicheStack services, the tasks should call the function **OSTimeDly**.

Disabling Non-Essential NicheStack Modules

Because the NicheStack networking stack is highly configurable, it is possible to include many modules (for example, FTP client/server, web server, and so forth). Every module included in your system may result in some performance degradation due to the overhead associated with having the Nios II processor service these modules.

This degradation can happen because the main NicheStack task, **tk_netmain** periodically polls each of these modules. Also, these modules may insert time-based, call back functions, which further decrease the overall performance of the networking stack.

You can control what is enabled or disabled in the NicheStack networking stack through a series of macro definitions in the **ipport.h** configuration file. Additionally, some settings are inserted in the Nios II processor's **system.h** file through the NicheStack's software component GUI. A list of NicheStack features and modules to disable, which may increase system performance, follows. (To disable a particular feature or module, ensure that its `#define` statement is not present in **ipport.h** or the **system.h** configuration files.)

The NicheStack features to disable include the following:

- **IN_MENU**S (enable NicheTool command interface)
- **NPDEBUG** (enable debugging aids)
- **MEM_WRAPPERS** (debugging aid to validate memory)
- **QUEUE_CHECKING** (debugging aid to validate memory queues)
- **MULTI_HOMED** (not needed if only one networking device)
- **IP_ROUTING** (not needed if only one networking device)

The NicheStack modules to disable include the following:

- **PING_APP** (enable ping support)
- **UDPSTEST**, **TCP_ECHOTEST** (enable echotest programs)
- **FTP_CLIENT**, **FTP_SERVER** (enable ftp client/server)
- **TELNET_SVR** (enable Telnet server)
- **USE_SYSLOG_TASK** (enable statistics collection)
- **SMTP_ALERTS** (enable email client)
- **INCLUDE_SNMP** (enable SNMP server)
- **DNS_SERVER** (enable DNS server)



Design Tip: Disabling unused NicheStack networking stack features and modules in your system helps increase overall system performance.



The NicheStack networking stack also supports a wide variety of features and modules not listed here. Refer to the NicheStack documentation and your **ipport.h** file for more information.

Using Faster Packet Memory

The performance of the NicheStack networking stack can be increased by using fast, low-latency memory for storing Ethernet packets. This section describes this optimization and explains how it works.

Background

The NicheStack networking stack uses a memory queue to assemble and receive network packets. To send a packet, the NicheStack removes a free memory buffer from the queue, assembles the packet data into it, and passes this buffer memory location to the Ethernet device driver. To receive the data, the Ethernet device driver removes a free memory buffer, loads it with the received packet, and passes it back to the networking stack for processing. The NicheStack networking stack allows you to specify where its queue of buffer memory is located and how this memory allocation is implemented.

By default, the Altera version of the NicheStack networking stack allocates this pool of buffer memory using a series of `calloc` function calls that use the system's heap memory. Depending on the design of the system, and where the Nios II system memory is located, this could impact overall system performance. A potential scenario in which this could occur is in cases where your Nios II processor's heap segment has been placed in high latency or slow memory.

Additionally, in the case where the Ethernet device utilizes DMA hardware to move the packets and the Nios II processor is not directly involved in transmitting or receiving the packet data, then this buffer memory must exist in an "uncached" region. This further degrades the performance because the Nios II processor's data cache is not able to offset any performance issues due to the "slow" memory.

The solution is to use the fastest memory possible for the networking stacks buffer memory, preferably a separate memory not used by the Nios II processor for programmatic execution.

Solution

The **ipport.h** file defines a series of macros for allocating and deallocating big and small networking buffers. The macro names begin with `BB_` (for "big buffer") and `LB_` (for "little buffer"). Following is the block of macros with the definitions in place for TSE driver support.

```
#define BB_ALLOC(size)      ncpalloc(size)
#define BB_FREE(ptr)       ncpfree(ptr)
#define LB_ALLOC(size)     ncpalloc(size)
#define LB_FREE(ptr)       ncpfree(ptr)
```

You can use these macros to allocate/deallocate memory any way you choose. In the case of the example design that accompanies this application note, these macros are redefined to allocate memory from MRAM memory (a fast memory structure inside the FPGA). This faster memory resulted in a 4% to 45% performance increase, depending on the system.



The Altera version of NicheStack does not use the BB_FREE or LB_FREE function calls. Therefore, any memory allocated via the BB_ALLOC and LB_ALLOC function calls occurs at run time, and is never freed.



Design Tip: Using fast, low latency memory for NicheStack's packet storage can improve the overall performance of the system.

Accelerating the Packet Checksum

The network checksum is a critical bottleneck to increasing the overall networking performance of the system. However, by using the Altera C2H compiler to accelerate the network checksum, you can increase the the system's networking performance.

Background

Ethernet networks use a checksum routine for guaranteeing the validity of transmitted data. This checksum is applied to the IP header, and is also used by the ICMP, IGMP, UDP, and TCP protocols for their own data headers and data.

The checksum operates by taking the 1's complement sum of the data octets of the packet (including the checksum field), where each octet is paired to form a 16-bit operand. When data is transmitted, the checksum field is set to all 0's, the 1's complement sum is taken of all the 16-bit coupled octets, and the 1's complement of the resultant value is stored in the checksum field. When packet data is received, however, the 1's complement sum is taken of all the 16-bit coupled octets (including the checksum field). If the result is equal to all 0's, the packet is valid.

While the algorithmic "work" being performed by this checksum does not seem to be very computationally intensive, the effect of running this checksum on every sent or received packet and their respective protocol data sections, can have the aggregate effect of degrading overall

networking performance. Because of this, most checksum routines are often written in “hand optimized” assembly code, which is the case in the NicheStack networking stack. However, further performance gains can be achieved by accelerating the checksum algorithm with FPGA hardware resources.

Optimizing the Packet Checksum

In the NicheStack networking stack, the checksum routine is configurable by setting a macro in the `ipport.h` configuration file, as follows:

```
#define cksum <function you want to call for the checksum>
```

You can set this macro to install any checksum implementation you want.

However, Altera's version of the NicheStack networking stack contains additional source code to enable three different checksums for experimentation and benchmarking (C source, Nios II assembly, and “hooks” for a C2H hardware checksum). More information, including detailed instructions, about how to create and use the C2H hardware checksum can be found in the `readme.doc` file, present in the example design zip file that accompanies this document.

In this application note, a C source code implementation of the network checksum has been optimized using Altera's C2H compiler. The results for accelerating the checksum via C2H can be found in the benchmark results table ([Table 4 on page 22](#)). In most cases, a C2H-optimized checksum routine yields a 6% to 40% performance improvement over the optimized assembly routine, depending on the configuration of the system.



Design Tip: Accelerating the performance of the network checksum routine, via dedicated hardware resources on the FPGA, can greatly accelerate overall network performance.

“Super Loop” Mode

Although the Altera-supported version of the NicheStack networking stack requires MicroC/OS-II for its operation, the stack can be enabled to run without an operating system. In this mode of operation, MicroC/OS-II is replaced with a single, never-ending software loop that services the stack and runs the user application.

Removing the MicroC/OS-II operating system from your system can result in slightly higher networking performance, but this comes at the expense of additional complexity in the software design of your system. It can be very easy to create “pathological” systems where your

application code consumes all of the processor's time, and without frequent calls to a stack servicing function, the effective networking performance deteriorates.

This application note does not attempt to benchmark the "Super Loop" system, but it is mentioned here as another possible area of optimization. General details of how to create the "Super Loop" system can be found in the NicheStack reference manuals (mentioned in the ["Appendix" on page 25](#)).



Design Tip: Although not officially supported by Altera, the NicheStack networking stack can be utilized without the MicroC/OS-II operating system. Doing so may provide additional networking performance benefits.

Ethernet Device

An important parameter in the total performance of your Ethernet application is the function and capabilities of the network interface device itself. Because the function of this device is to translate the physical Ethernet packets into datagrams that can be accessed by the stack, its performance is critical to the overall performance of your networking application.

Link Speed

For most embedded networking applications, the network physical layer is composed of either 100BASE-TX or 1000BASE-T Ethernet, which uses twisted copper wires for the transport medium. The maximum data transport rate (in one direction) for 100BASE-TX is 100 Mbits/sec, while 1000BASE-T can accommodate 1000 Mbits/sec.

It is very difficult for an embedded networking device to completely use a 100 Mbit link (much less a 1000 Mbit link), but a faster link provides better performance most of the time. This is because the 1000 Mbit link has a larger overall carrying capacity for data. The improvement is especially noticeable in cases where the link is shared among several different devices that are using the link simultaneously.

Selecting the Right Hardware

Two supported Ethernet device solutions present in the Nios II Embedded Development are the LAN91C111 (by SMSC) and the Altera TSE MAC. Apart from the obvious difference in supported Ethernet speeds, with the LAN91C111 supporting 10/100 Mbit and the TSE Ethernet MAC supporting 10/100/1000 Mbit networks, both devices have radically different implementations that impact networking performance.

Network Interface Comparison (LAN91C111 vs. Altera TSE MAC)

Both the LAN91C111 and Altera TSE MAC essentially perform the same role, that is, to translate an application's Ethernet data into physical bits on the Ethernet link. However, as seen in [Table 2](#), there are some key differences between them that can impact network performance.

Parameter	Ethernet Device	
	LAN91C111	Altera TSE MAC
Type	external chip	FPGA IP
Control Interface	Avalon MM (Tri-state Bridge)	Avalon MM
Data Interface	Avalon MM (Tri-state Bridge)	Avalon ST
Data Width (bits)	8,16, 32	8, 32
Supported Link Speeds (Mbits/sec)	10/100	10/100/1000
Recv FIFO Depth	8 KB for send/recv (combined)	64 Bytes to 256 Kbytes
Send FIFO Depth	8 KB for send/recv (combined)	64 Bytes to 256 Kbytes
DMA	None	Altera SGDMA (required)
PHY Interface (Integrated)	100BASE-TX/10BASE-T	None
PHY Interface (External)	MII (100 Mbits/sec)	MII (100 Mbits/sec), GMII (1000 Mbits/sec)

In terms of Ethernet link speed, the LAN91C111 only supports up to a 100 Mbit link speed, while the TSE MAC can support up to a 1000 Mbit link speed. Additionally, the TSE MAC is capable of sending and receiving Ethernet data more quickly than the LAN91C111 because of the SGDMA peripherals. Finally, the TSE MAC provides a greater range of send and receive FIFO depth to be selected, while the LAN91C111 restricts you to a fixed 8 Kbyte memory space shared by both the send and receive operations.

NicheStack Device Driver Model

The NicheStack networking stack presents users with a simplified device driver model for integrating their Ethernet devices, and both the LAN91C111 and Altera TSE MAC solutions have been fully optimized to support this model.

In the LAN91C111 device driver, the Nios II CPU is solely responsible for performing the movement of Ethernet packet data to and from the device. However, in the TSE MAC device driver, the SGDMA peripherals are responsible for the movement of the Ethernet packet data to and from the TSE MAC.

These SGDMA peripherals can operate much more efficiently than the Nios II processor for data movement operations (on a per clock basis), and therefore results in an overall performance increase in the system. The benchmark results show a performance increase of more than 15% to 60% of the TSE MAC solution to the LAN91C111 solution for this very reason.

Benchmarking Results and Analysis

The previous sections have described several optimizations that can be used to increase the performance of a networking system. This section describes a method to evaluate the effectiveness of each one. The best way to evaluate the optimizations is to use a benchmarking application that measures the impact of applying each optimization.

Overview

A simple benchmarking application was created to measure the overall networking performance. This application enables you to measure the Ethernet data transfer rate between two systems, such as a Nios II development board and a workstation using the TCP or UDP protocols.

During a benchmarking test, one machine assumes the role of the “sender” and the other machine becomes the “receiver.” The sender opens a connection to the receiver, transmits a specified amount of data, and prints out a throughput measurement in Mbits/sec. Likewise, the receiver waits for a connection from the sender, begins receiving Ethernet data, and at the end of the data transmission prints out the total throughput, in Mbits/sec.

The benchmarking application has been structured to be as simple as possible. Both the sender and receiver parts of the program perform no additional work apart from sending and receiving Ethernet data. Additionally, for standardization purposes, all network operations use the industry standard Sockets API in their implementation.



More information about the benchmarking program, including detailed information about how to build and operate it, can be found in the **readme.doc** file in the example design file that accompanies this application note.

Test Setup

The benchmarking tests were conducted between a workstation and a Nios II development board. The workstation used was a Dell Optiplex GX280 workstation running the Windows XP Professional operating system, with two Pentium 4 (3.2GHz) CPUs. The Nios II development board used was a Nios II Stratix® II RoHS development board with a Marvell PHY 10/100/1000 daughter card. The workstation was lightly loaded, meaning that the only user applications running were the benchmark program and the Nios II IDE.

The Ethernet connection used between the two systems was a single twisted pair, networking cable used in a direct-connection fashion (no hub or switch was used between the connections).

Finally, to minimize the impact of spurious network communications on the Ethernet link, only the TCP/IP protocol suite was enabled for the network link on the Windows workstation; all other networking protocols and applications were disabled.

Test Systems

The benchmarking analysis was structured to demonstrate how changing key parameters in an Ethernet system can lead to radical performance changes. The analysis compared two of Altera's Ethernet networking solutions:

- A system using the LAN91C111 chip (10/100 Mbit)
- A system using the Altera Triple Speed Ethernet (TSE) MAC (10/100/1000 Mbit), with a Marvell PHY

The goal of this benchmark analysis is to highlight the performance improvements that can result from using an optimized Ethernet device that utilizes SGDMA channels to handle the data movement. The tests also measure the relative performance differences of using the Nios II processor with SSRAM and DDR SDRAM.



Test runs for the LAN91C111 were conducted on a 100 Mbit link, because this is the maximum link speed supported by the LAN91C111 device.

This benchmark test examines the merits of applying various optimizations to both the Nios II processor and the NicheStack networking stack. The first parameter tested is the effect of doubling the instruction and data cache sizes for the processor. The second parameter tested is the effect of increasing the Nios II processor's clock frequency

from 83.333 MHz to 150 MHz. All the tests also measure the relative performance differences of operating the Nios II processor with both SSRAM and DDR SDRAM.

The effect of applying various hardware optimizations to the NicheStack networking stack were also measured. These included the use of a hardware checksum (generated via the C2H compiler), the use of fast internal memory for packet storage, using a combination of the two, and not using either.

Test Methodology

This section describes the parameters used in the benchmarking tests.

Ethernet Link Type

The Ethernet link selected for connecting the workstation to the Nios II board used a single 100/1000 Mbit cable in a point-to-point configuration (no hub or switch). This choice mitigated any effects an additional piece of networking hardware might have had in the test system.

In most networking applications, however, your system may be connected to another host through one (or more) Ethernet hubs or switches. These extra connections may increase the communication latency. Therefore, the benchmark numbers presented here should be viewed as the “idealized” performance of a (near) perfect Ethernet connection.

Protocols Tested

All benchmark operations were conducted using the the TCP protocol, because the TCP protocol guarantees that all data sent by the “transmitter” arrives at the “receiver.” This means that the throughput numbers reported are legitimate.

The benchmark application can measure UDP transmission speeds, but does so without accounting for lost or missing Ethernet packets. Therefore, the UDP test only measures the speed at which the transmitter can send all of the data using the UDP protocol, without considering whether the data made it to the receiver at all.

Data Transmission Sizes

For this series of tests, a total data size of 100 megabytes (100,000,000 bytes) was selected. This data size was chosen to increase the total amount of time spent in the course of the test, to more clearly capture the average performance of both the sender and receiver.

Furthermore, the largest TCP payload size was used for Ethernet packet transmission (1458 bytes). This payload size was chosen to provide an upper bound of Ethernet performance, that is, the best expected performance numbers achievable in the design.



Because the benchmarking application uses the Sockets API, the payload size (1458 bytes) directly maps to the length parameter in the `send` (TCP) and `sendto` (UDP) function calls. Following is an example of a `send` function call in TCP:

```
send(int socket, const void*buffer, size_t length, int flags);
```

Test Runs

For every Nios II configuration, the data transmission time and average data throughput was measured with the Nios II system as both the sender and the receiver. Three consecutive measurements were taken and the average of these runs was recorded as the final measurement.

Nios II System Software Configuration

The benchmark application uses Altera's recommended structure for Nios II NicheStack-based applications. The application relies on the MicroC/OS-II and NicheStack Sockets API for operation. The following configurations, were applied to all test systems.

NicheStack Networking Stack Configuration

The NicheStack networking stack was built by selecting the default configuration in the configuration wizard. This configuration provides a minimal set of general purpose functionality to enabled networking operations using the TCP and UDP protocols.

Additionally, the following MicroC/OS-II thread priorities were selected for the two core NicheStack tasks:

- `tk_netmain` = priority 2
- `tk_nettick` = priority 3

MicroC/OS-II Configuration

The default MicroC/OS-II configuration was also selected for the operation of the networking stack. This configuration provides all the basic MicroC/OS-II services.

Benchmark Application

The benchmark application uses the Sockets API. The configuration for the application is as follows:

- benchmark application = priority 4
- benchmark initialization thread = priority 1



More information on the benchmark application and its operation can be found in the design files that accompany this application note.

General Application and System Library Settings

Both the benchmark application and the associated system library were compiled using the Nios II GNU tool chain with the `-O3` optimization enabled. In the test cases that involve changes to the run-time memory (either SSRAM or DDR SDRAM), the entire memory was selected for the application's binary segments (Text, Data, BSS, and so forth).

Workstation System Software

The workstation benchmark application was compiled using the GNU tool chain for the Cygwin environment, targeting the x86 architecture. Because the workstation benchmark application re-utilizes much of the same source code base as the Nios II application, it uses the Sockets API for conducting this test.

Nios II Test Hardware

Based on the performance guidelines for the application, networking stack, and Networking device peripherals, the following four systems were created for benchmarking. [Table 3](#) summarizes the results.

Table 3. Ethernet Benchmark Test Matrix *Note (1)*

System	MHz	Nios II Cache (Kbytes)		Memory		Optimizations				Link Speed (Mbps/Sec)	
		Inst.	Data	SSRAM	DDR SDRAM	None	HW	FM	HW/FM	100	1000
LAN91C111	83.333	4	2	x	x	x	x	x	x	x	
TSE Standard	83.333	4	2	x	x	x	x	x	x		x
TSE Big Cache	83.333	8	8	x	x	x	x	x	x		x
TSE Optimized	150	8	8	x	x	x	x	x	x		x

Note to Table 3:

- (1) HW = Hardware checksum optimization
 FM = Fast packet memory optimization
 x = Feature is present

- **LAN91C111**—This system is the Nios II, Stratix II RoHS “full featured” reference design that is included in the Nios II development kit. The design was modified to run at 83.333 MHz to match the other systems being benchmarked. This design includes an interface to the LAN91C111 MAC/PHY chip, and memory interfaces to DDR SDRAM and SSRAM memory.
- **TSE Standard**—This system is the Nios II, Stratix II RoHS “TSE_SGDMA” reference design that is included in the Nios II development kit. This design includes an interface to the TSE MAC, driven by two separate SGDMA engines. Memory interfaces are provided to DDR SDRAM and SSRAM memory. The core system operates at a clock frequency of 83.333 MHz, with an additional clock driving the MAC component at 125 MHz. This system also has the fast packet memory and C2H hardware checksum optimization.
- **TSE Big Cache**—This is the same design as TSE Standard, but includes larger instruction and data cache memories for the Nios II processor. The instruction cache memory is 8 Kbytes (vs. 4 Kbytes in the TSE Standard system) and the data cache memory is 8 Kbytes (vs. 2 Kbytes in the TSE Standard system).

- **TSE Optimized**—This system is almost the same design as TSE Big Cache, but the overall system clock frequency for the system has been increased to 150 MHz. This clock increase was made possible by using a pipeline bridge peripheral to separate the faster peripherals from the slower peripherals in the system.



The fourth design (TSE Optimized) is included in the design files that accompany this application note.

Test Results

Table 4 shows the results from the benchmark testing.

System Name	Test Type	Configuration							
		DDR SDRAM				SSRAM			
		None	FM	HW	FM/HW	None	FM	HW	FM/HW
LAN91C111	TX	24.183	25.845	23.472	27.924	27.711	28.888	30.114	32.262
	RX	19.840	21.161	17.729	22.903	21.530	23.221	22.854	24.667
TSE Standard	TX	27.862	31.800	32.524	33.916	41.395	43.828	50.332	54.462
	RX	19.066	25.453	20.889	27.970	27.712	32.217	31.608	35.692
TSE Big Cache	TX	36.172	44.930	46.179	47.703	58.740	62.898	76.260	82.118
	RX	21.805	33.274	25.912	38.047	38.059	42.662	43.994	49.520
TSE Optimized	TX	58.708	71.259	82.778	80.702	83.827	94.742	107.627	108.868
	RX	36.019	53.076	41.619	59.682	54.485	65.451	62.628	74.569

Note to Table 4:

- (1) All figures are in Mbits/sec
 None = Neither optimization is present
 FM = Fast packet memory optimization
 HW = Hardware checksum optimization

Analysis

The test results show that the Nios II Ethernet transmission speed (TX) is much greater than the receive speed (RX). This result is most likely because the computational overhead of transmitting packets tends to be much less than that for receiving packets.

The results from the LAN91C111 system show that its throughput is constrained both computationally and in terms of data movement, because the Nios II processor is involved in both operations. Therefore, when the latency of the memory (moving from DDR SDRAM to SSRAM memory) is decreased, the performance increases in a fairly linear way.

Additionally, both the fast packet memory and C2H checksum optimizations improved the overall performance, but the main bottleneck is still the Nios II processor's ability to move data in the system. The Nios II data movement operations, that is, moving packet data from the LAN91C111 device to memory, were clearly the slowest operations in the whole system. Any major performance improvements in this system are only going to happen by improving the data movement operations of the packets to and from the Ethernet device.



The LAN91C111 device can be used with an Altera Avalon DMA peripheral, but this implementation is beyond the scope of this application note.

The TSE Standard system results are interesting because the data movement operations between the networking stack and the Ethernet device are accomplished via the use of the SGDMA engines instead of the Nios II processor. There is a significant performance improvement in both sending and receiving Ethernet data (48% increase in TX and 69% increase in RX) when the high latency memory (DDR SDRAM) is replaced by the lower latency memory (SSRAM). These improvements are further magnified when the fast packet memory and hardware checksum optimizations are applied to the DDR SDRAM and SSRAM based systems (61% TX improvement and 28% RX improvement).

The results of the TSE Big Cache system show the importance of increasing the cache memory in the TSE systems. By increasing the instruction cache from 4 Kbytes to 8 Kbytes, and increasing the data cache from 2 Kbytes to 8 Kbytes, there is a big performance increase in both the TX and RX cases. With the system running in DDR SDRAM, the TX performance jumps from about 28 Mbits/sec to 36 Mbits/sec (30% increase) and the RX performance jumps from 19 Mbits/sec to about 22 Mbits/sec (14% increase). Moreover, in SSRAM the TX performance jumps from 41 Mbits/sec to 59 Mbits/sec (42% increase) and the RX performance jumps from 27 Mbits/sec to 38 Mbits/sec (37% increase). The test results show that the increase in data cache sizes helped to increase the overall computational performance of the Nios II processor, which in turn increased the overall throughput of the networking stack. Additionally, the performance increase of the fast packet memory and hardware checksum served to further boost the performance as well, with both the TX performance and RX performance increasing substantially over the standard systems.

Increasing the overall frequency of the entire system (Nios II processor, Ethernet device, and memory) also had a profound effect on the overall networking performance of the system. In the TSE Optimized design, the frequency of the system was increased from 83.333 MHz to 150 MHz, an 80% increase in MHz from the TSE Standard system. This change resulted

in approximately a 60% increase in TX and RX performance in the DDR SDRAM system, and about a 40% increase in TX and RX performance in the SSRAM system, respectively. Additionally, both the fast packet memory and hardware checksum optimizations also seemed to maintain this overall performance increase as well. The TSE Optimized system running from SSRAM, then, was able to achieve 107 Mbits/sec TX and 63 Mbits/sec RX performance, about a 100% increase in performance over the TSE Standard system.

Note that applying both the fast packet memory and C2H hardware checksum optimizations to a given system tends to provide a greater level of performance than applying each optimization by itself. This makes sense because the C2H hardware checksum operates more quickly on faster memory, which is the whole point of the fast packet memory optimization.

Finally, note that in the some test cases where the DDR SDRAM was used for the Nios II processor's memory, the C2H hardware checksum yielded results that were sub-optimal (the performance was less than using the standard assembly checksum). This result was most likely due to bank switching caused by the accelerator's access of the DDR SDRAM memory; however, this problem went away when the fast packet memory was used in conjunction with the C2H hardware checksum.

Conclusion

As seen in the empirical benchmark results, minor performance increases in your Ethernet system can be obtained by applying a single hardware optimization; however, achieving significant Ethernet performance increases involves applying several hardware optimizations together in the same system.

In decreasing order of importance, the optimizations you should consider for their Ethernet system are as follows:

- DMA engine for moving data to and from the Ethernet device
- Increasing the overall system frequency (CPU, DMAs, memory, and so forth)
- Using low-latency memory for Nios II execution
- Using C2H to accelerate the network checksum
- Using fast packet memory to store Ethernet data

Finally, the overall performance you seek from your Ethernet application depends on the nature of the application itself. This application note has provided you with general techniques to accelerate Nios II Ethernet applications, but the final measure of success is whether your application meets the performance goals you established.

Appendix

General Information for TCP/IP Networking

The following resources were used in the construction of this application note, and can provide you with more information regarding Ethernet, the TCP/IP protocol, and the Sockets API:

General Information:

- Richard Stevens, *UNIX Network programming*
- Douglas Comer, *Internetworking with TCP/IP volume 3*
- General Ethernet Information (en.wikipedia.org/wiki/Ethernet)

Additionally, more information regarding Altera's tools and technology can be found on the Altera literature page (www.altera.com/literature).

NicheStack Documentation



For more information about using “Super Loop” mode, the “Zero Copy” API, and so forth, refer to the NicheStack TCP/IP Stack documentation in the **NicheStackRef.zip** file located in the `<Nios II EDS install path>/components/altera_niche/UCOSII/src/downloads/packages` directory.

Additional NicheStack Information

The NicheStack TCP/IP Networking stack is a software library licensed by Altera from InterNiche Technologies. If you are interested in licensing the NicheStack networking stack for use in your Nios II application, check the terms and conditions here: www.altera.com/nichestack.

The version of the NicheStack networking stack distributed by Altera provides you with basic TCP/IP networking functionality. If your application requires additional application modules, or protocol support, visit the InterNiche website for more information: www.iniche.com.

Additional Network Technology Solutions

The device driver support included in the Altera version of the NicheStack networking stack supports both the LAN91C111 MAC/PHY chip and Altera TSE MAC IP. Additional networking device IP is available at Altera's IP Megastore: www.altera.com/products/ip/ipm-index.html

Document Revision History

Table 5 shows the revision history for this application note.

Date and Document Version	Changes Made	Summary of Changes
May 2007 v1.0	Initial release.	—



101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support:
www.altera.com/support/
Literature Services:
literature@altera.com

Copyright © 2007 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



I.S. EN ISO 9001