# Accelerating Nios II Networking Applications

AN-440-2.0 Application Note

This application note describes key optimizations you can use to accelerate the performance of your Nios® II networking application. In addition, this document describes how the different parts of a Nios II Ethernet-enabled system work together, how the interaction of these parts corresponds to the total networking performance of the system, and how to benchmark the system.

Ethernet is a standard data transport paradigm for embedded systems across all applications because it is cheap, abundant, mature, and reliable.

## Downloading the Ethernet Acceleration Design Example

The Nios II ethernet acceleration design example is an integral part of this application note. The design example shows how these acceleration techniques can be applied in a real working Nios II system. The **readme.doc** file, located in the design example folder, provides additional hands-on instructions demonstrating how to implement these acceleration techniques in a Nios II system. The **readme.doc** file also provides performance benchmark results.

You can find the Nios II ethernet acceleration design example on the Nios II Ethernet Acceleration Design Example page of the Altera website.

Download the design example file, and unzip the file into a working directory.

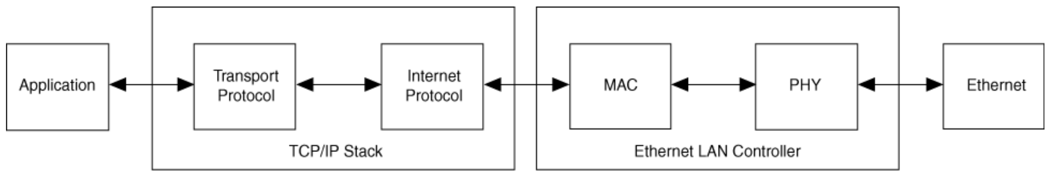
## The Structure of Networking Applications

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**



Subscribe

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, such as controlling the speed of a motor or providing the UI for an embedded kiosk. The networking stack provides the application with an application programming interface (API), usually Sockets, 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 through 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, such as address resolution protocol (ARP), transmission control protocol (TCP), and user datagram protocol (UDP). These stacks generally require a significant amount of processing power.

The stack uses the Ethernet device 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.

You must consider the link layer, or physical media over which the Ethernet datagrams travel, when constructing a network enabled system. Depending on the location of the embedded system, the Ethernet datagrams might traverse a wide variety of physical links, such as 10/100 Mbit twisted pair and fiber optic. Additionally, the datagrams might experience latency if they traverse long distances or need to pass through many network switches in order to arrive at their destination.

## Relationships Between Networking System 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 method provides you with an upper bound for 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

A wide variety of tools is 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 the following tools:

- GNU Profiler
- Timer Peripheral IP Core
- Performance Counter IP Core

This application note does not explore the use of these tools or how they were applied to find networking bottlenecks in the system. 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 performed. 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 executes on the same processor as the network stack, and competes with it 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. This technique has a twofold benefit. 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 processor.

## Software Optimizations

- **Compiler Optimization Level**—Compile your application with the highest compiler optimization possible. Higher optimizations result in denser, faster code, 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. Balance the application's priority levels against the priority levels assigned to the NicheStack's core tasks, discussed in "Structure of the NicheStack Networking Stack" on page 7.

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

## Hardware Optimizations

- **Processor Performance**—You can increase the performance of the Nios II processor in the following ways:
  - **Computational Efficiency**—Selecting the most computationally efficient Nios II processor core is the quickest way to improve overall application performance. The following Nios II processor cores are available, in decreasing order of performance:
    - Nios II/f—optimized for speed
    - Nios II/s—balances speed against usage of on-chip resources
    - Nios II/e—conserves on-chip resources at the expense of speed
  - **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 increases the processor's performance by allowing the Nios II processor to perform more transactions to the memory before another Avalon master port can assume control of the memory.
  - **Instruction and 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, such as SDRAM or double data rate (DDR) SDRAM. 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, ensure that the processor's execution memory is in 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 a FIFO bridge IP core 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, included Nios II ethernet acceleration design example, 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 most common ways to accelerate application level algorithms are as follows:
  - **Custom Instruction**—Offload the Nios II processor by using hardware to implement a custom instruction.
  - **Custom Peripheral**—Create a block of hardware that performs a specific algorithmic task, as a peripheral controlled by the Nios II processor.

For more information about hardware optimizations, refer to the *Avalon Memory-Mapped Design Optimizations* and *Hardware Acceleration and Coprocessing* chapters of the Embedded Design Handbook.

## The Sockets API

After tuning your application to become more computationally efficient (thereby freeing more of the processor's time for operating the networking stack), you can optimize how the 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 and UDP. 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 | Connectionless | 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 multimedia 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 and Receive 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 and receive 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()`. For 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. Combine data that you want to send (or receive) and call these functions with the largest possible amount of data at one time.

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

- **Minimize latency when sending data**—Although the TCP Sockets `send()` function can accept an arbitrary number of bytes, those bytes might not be immediately sent as a packet. This situation is especially likely 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. Small data chunks are coalesced 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.

  Setting a socket's `TCP_NO_DELAY` flag, with the `setsockopt()` function call, disables the Nagle Algorithm. The socket immediately sends whatever bytes are passed in as a TCP packet. Disabling the Nagle Algorithm 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. You can find an example of setting the `TCP_NO_DELAY` flag in the benchmarking application software in the Nios II ethernet acceleration design example.

☞ While disabling the Nagle Algorithm usually causes smaller packets to be immediately sent over the network, the networking stack might still coalesce some of the packets into larger packets. This situation is especially likely in the case of the Windows workstation platform. However, you can expect the networking stack to do so with much lower frequency than if the Nagle Algorithm were 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 application manages the send and receive data buffers directly, eliminating an extra level of data copying performed by the Nios II processor.

This application note does is not discuss details of performance optimization with the Zero Copy API. Refer to the "Appendix" on page 18 for pointers to more information.

☞ **Design Tip:** Using the NicheStack Zero Copy API can 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 Embedded Design Suite (EDS) is optimized for use with the MicroC/OS-II (RTOS), and includes device driver support for the Altera® Triple Speed Ethernet MegaCore function, which serves as the media access control (MAC).

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

This section describes the targeted optimizations that you can use to increase the performance of the NicheStack networking stack directly.

### 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 timekeeping, 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 provide the best networking performance possible for your system. However, in your embedded system you might need to override these priorities because your application task (or tasks) run more frequently than these tasks. Overriding these priorities, however, might result in performance degradation of network operations, as the NicheStack networking stack has fewer 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, the MicroC/OS-II scheduler places your application task from a running state into a waiting state. The scheduler 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 do not use the NicheStack services, the tasks should call the function `OSTimeDly()`.

### Disabling Nonessential NicheStack Modules

Because the NicheStack networking stack is highly configurable, many modules are available for you to optionally include. Some examples are an FTP client, an FTP server, and a web server. Every module included in your system might 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 might insert time-based callback 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. In addition, the NicheStack software component inserts some definitions in the **system.h** file belonging to the board support package (BSP). A list of NicheStack features and modules to disable, which can increase system performance, follows. (To disable a particular feature or module, ensure that its `#define` statement is present in neither the **ipport.h** file nor the **system.h** configuration file.)

The NicheStack features to disable include the following:

- IN_MENUS—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 simple network management protocol (SNMP) server
- DNS_SERVER—enable domain name system (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

You can increase the performance of the NicheStack networking stack 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 allocation method could impact overall system performance. For example, if your Nios II processor's heap segment is in high latency or slow memory, this allocation method might degrade performance.

Additionally, in the case where the Ethernet device utilizes direct memory access (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. Lack of buffer caching 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 Triple Speed Ethernet device 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 and deallocate memory any way you choose. The Nios II ethernet acceleration design example redefines these macros to allocate memory from MRAM memory (a fast memory structure inside the FPGA). This faster memory results in various degrees of performance increase, depending on the system. For detailed performance improvement figures, please refer to the **readme.doc** file included in the design example.

☞ The Altera version of NicheStack does not use the `BB_FREE()` or `LB_FREE()` function calls. Therefore, any memory allocated with the `BB_ALLOC()` and `LB_ALLOC()` function calls is allocated 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 a custom hardware peripheral to accelerate the network checksum, you can increase 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 Internet control message protocol (ICMP), Internet group management protocol (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 algorithm performed by this checksum does not appear 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 potential degradation, checksum routines are often written in hand-optimized assembly code, as they are in the NicheStack networking stack. However, you can achieve further performance gains by accelerating the checksum algorithm implementation with a custom checksum hardware peripheral.

### Optimizing the Packet Checksum

In the NicheStack networking stack, you can configure the checksum routine 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 language, and hooks for a custom hardware checksum peripheral). You can find more information about how to incorporate a custom hardware checksum peripheral in the **readme.doc** file, including detailed instructions. **readme.doc** is in the Nios II ethernet acceleration design example.

☞ **Design Tip:** Accelerating the performance of the network checksum routine, using 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, you can configure the stack to run without an operating system. In this mode of operation, MicroC/OS-II is replaced by an infinite 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 improvement comes at the expense of additional complexity in the software design of your system. It is 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.

Although the Super Loop system is another possible method of optimization, this application note does not attempt to benchmark it. You can find information about how to create a Super Loop system in the NicheStack reference manuals (mentioned in the "Appendix" on page 18).

☞ **Design Tip:** You can use the NicheStack networking stack without the MicroC/OS-II operating system. Doing so can provide additional networking performance benefits. However, Altera does not support this configuration.

## Ethernet Device

An important parameter in the total performance of your Ethernet application is the function and capabilities of the network interface device itself. The function of this device is to translate the physical Ethernet packets into datagrams that can be accessed by the stack. Therefore 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. However, a faster link provides better performance most of the time, because the 1000 Mbit link has a larger overall carrying capacity for data. The improvement is especially noticeable in cases where several different devices share the link and use it simultaneously.

## Network Interface (Altera Triple Speed Ethernet MegaCore Function)

The Nios II EDS supports the Altera Triple Speed Ethernet MegaCore function. The Triple Speed Ethernet MegaCore function's role is essentially to translate an application's Ethernet data into physical bits on the Ethernet link. The Triple Speed Ethernet MegaCore function supports 10/100/1000 Mbit networks. Table 2 lists the key design parameters that impact network performance.

**Table 2. Triple Speed Ethernet MegaCore Function**

| Parameter | Altera Triple Speed Ethernet MegaCore Function |
|---|---|
| Type | FPGA IP |
| Control Interface | Avalon-MM |
| Data Interface | Avalon-ST |
| Data Width (bits) | 8, 32 |
| Supported Link Speeds (Mbits/sec) | 10/100/1000 |
| Recv FIFO Depth | 64 Bytes to 256 Kbytes |
| Send FIFO Depth | 64 Bytes to 256 Kbytes |
| DMA | Altera Scatter-Gather DMA (required) |
| PHY Interface (Integrated) | None |
| PHY Interface (External) | MII (100 Mbits/sec),<br>GMII (1000 Mbits/sec) |

The Triple Speed Ethernet MegaCore function is capable of sending and receiving Ethernet data quickly because of the Scatter-Gather DMA peripherals. The Triple Speed Ethernet MegaCore function also allows you to select from a flexible range of send and receive FIFO depths.

## NicheStack Device Driver Model

The NicheStack networking stack presents a simplified device driver model for integrating Ethernet devices, and the Altera Triple Speed Ethernet MegaCore function solution is fully optimized to support this model.

In the Triple Speed Ethernet MegaCore function device driver, the Scatter-Gather DMA peripherals are responsible for the movement of the Ethernet packet data to and from the Triple Speed Ethernet MegaCore function.

The Scatter-Gather DMA peripherals can operate much more efficiently than the Nios II processor for data movement operations (on a per clock basis), and therefore using the Triple Speed Ethernet MegaCore function device driver results in an overall performance increase in the system.

For information about the Triple Speed Ethernet MegaCore function, refer to the *Triple Speed Ethernet User Guide*. For information about the Scatter-Gather DMA peripheral, refer to the *Embedded Peripherals IP User Guide*.

# Benchmarking Setup, Results and Analysis

The previous sections have described several optimizations that you can use 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 measures the overall networking performance. This application enables you to measure the Ethernet data transfer rate between two systems, such as an Altera 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 the simplest possible structure. 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.

☞ You can find more information about the benchmarking program, including detailed information about how to build and operate it, in the **readme.doc** file in the Nios II ethernet acceleration design example.

## Test Setup

The benchmarking tests were conducted between a workstation and an Altera development board. The workstation used was a Dell Optiplex GX280 workstation running the Windows XP Professional operating system, with two Pentium 4 (3.2GHz) processors. The Altera development board used was a Stratix® IV GX development board. The workstation was lightly loaded, meaning that the only user applications running were the benchmark program and the Nios II Software Build Tools (SBT) for Eclipse.

The direct Ethernet connection between the two systems was implemented using a single twisted-pair networking cable.

### Test Systems

The benchmarking analysis demonstrates how changing key parameters in an Ethernet system can lead to radical performance changes.

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.

The test also measures the effect of applying various hardware optimizations to the NicheStack networking stack. These optimizations include the use of a hardware checksum (custom hardware peripheral), the use of fast internal memory for packet storage, and the use of a combination of these optimizations. The test makes measurements for these cases and for the case in which neither of the two listed optimizations is implemented.

## Test Methodology

This section describes the parameters used in the benchmarking tests.

### Ethernet Link Type

The Ethernet link selected to connect the workstation to the Nios II board uses a single 100/1000 Mbit cable in a point-to-point configuration (no hub or switch). This choice mitigates the potential effects of an additional piece of networking hardware on the test system.

In most networking applications, however, your system can be connected to another host through one (or more) Ethernet hubs or switches. These extra connections can increase the communication latency. The benchmark numbers present the idealized performance of an almost perfect Ethernet connection.

### Protocols Tested

All benchmark operations are conducted using the TCP protocol. The TCP protocol guarantees that all data sent by the transmitter arrives at the receiver, ensuring 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 arrived at the receiver.

### Data Transmission Sizes

This series of tests uses a total data size of 100 megabytes (100,000,000 bytes). This data size increases 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 tests use the largest TCP payload size for Ethernet packet transmission (1458 bytes). This payload size provides an upper bound of Ethernet performance, representing 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 test measures the data transmission time and average data throughput with the Nios II system as both the sender and the receiver. The tests take three consecutive measurements and record the average of these runs 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 is built with the default configuration. 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 is used 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

☞ You can find more information on the benchmark application and its operation in the Nios II ethernet acceleration design example.

### 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 `-03` optimization enabled. If the test cases involve any changes to the run-time memory, the entire memory would be selected for the application's binary segments, such as `.text`, `.data`, and `.bss`.

### Workstation System Software

The workstation benchmark application is compiled using the GNU tool chain for the Cygwin environment, targeting the x86 architecture. Because the workstation benchmark application reuses 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 and Test Results

For details regarding the Nios II test hardware and test results, see the **readme.doc** file included in the Nios II ethernet acceleration design example.

## Conclusion

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

Consider using the following optimizations for your Ethernet system, in decreasing order of importance:

- DMA engine for moving data to and from the Ethernet device
- Increasing the overall system frequency, including components such as the processor, DMAs, and memory
- Using low-latency memory for Nios II software execution
- Using a custom hardware peripheral 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 provides 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 establish.

# 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:
  - Comer, Douglas E., and Stevens, David L., *Internetworking With TCP/IP Volume III: Client-Server Programming and Applications, Linux/POSIX Socket Version*, Prentice Hall, 2000
  - Stevens, Richard, *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications*, Prentice Hall, 1999
  - Ibid., *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*, Prentice Hall, 1998
- You can find more information about Altera's tools and technology on the Literature and Technical Documentation page of the Altera website.

## NicheStack Documentation

For more information about using Super Loop mode and the Zero Copy API, refer to the NicheStack TCP/IP Stack documentation in the **doc31.zip** file located in the <*Nios II EDS install path*>/**components/altera_iniche/UCOSII/31src** 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 at the Nios II Networking Solutions page of the Altera website.

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 (www.iniche.com) for more information.

## Additional Network Technology Solutions

The device driver support included in the Altera version of the NicheStack networking stack supports the Altera Triple Speed Ethernet MegaCore function. Additional networking device IP is available on the Intellectual Property & Reference Designs page of the Altera website.

## Document Revision History

Table 3 shows the revision history for this document.

**Table 3. Document Revision History**

| Date | Version | Changes |
|---|---|---|
| August 2010 | 2.0 | ■ Update for Release 10.0<br>■ Remove C2H, replace with custom hardware peripheral component<br>■ Remove LAN91C111<br>■ Move test results table to the **readme.doc** file, included with the Nios II ethernet acceleration design example |
| June 2009 | 1.1 | Revised benchmarking data |
| May 2007 | 1.0 | Initial release. |