Exploring a Multithreaded Methodology to Implement a Network Communication Protocol on the Cyclops-64 Multithreaded Architecture

Ge Gan, Ziang Hu, Juan del Cuvillo, Guang R. Gao

University of Delaware
Dept. of Electrical and Computer Engineering
Newark, DE. 19716 USA
{gan,hu,jcuvillo,ggao}@capsl.udel.edu

Abstract

The IBM Cyclops-64 (C64) chip employs a multithreaded architecture that integrates a large number of hardware thread units on a single chip. A cellular supercomputer is being developed based on a 3D-mesh connection of the C64 chips. This paper introduces the Cyclops Datagram Protocol (CDP) developed for the C64 supercomputer system. CDP is inspired by the TCP/IP protocol, yet simpler and more compact. The implementation of CDP leverages the abundant hardware thread-level parallelism provided by the C64 multithreaded architecture.

The main contributions of this paper are: **(1)** *We have completed a design and implementation of CDP that is used as the fundamental communication infrastructure for the C64 supercomputer system.* **(2)** *CDP successfully exploits the massive thread-level parallelism provided on the C64 hardware, achieving good performance scalability;* **(3)** *CDP is quite efficient. Its peak throughput reaches 884Mbps on the Gigabit Ethernet, even it is running at the user-level on a single-processor Linux machine;* **(4)** *Extensive application test cases are passed and no reliability problems have been reported.*

1. Introduction

Cyclops-64 (C64) is a multithreaded architecture developed at the IBM T.J. Watson research center [4]. It is the latest version of the Cyclops cellular architecture that employs a unique multiprocessor-on-a-chip design [1] that integrates a large number of thread execution units, main memory banks, and communication hardware on a single chip. See Figure 1. The C64 chip, together with the host control log-
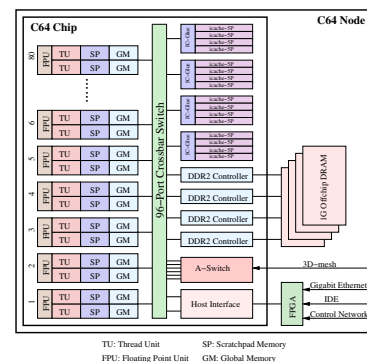
Figure 1. Cyclops-64 Node

ics and the off-chip memory, becomes the building block (i.e. the C64 node) of the C64 supercomputer system. See Figure 2. The C64 supercomputer system consists of tens of thousands of C64 nodes that are connected by the 3D-mesh network and the Gigabit Ethernet and can provide computing power at the Petaflops level.

To interconnect the two different subnetworks in the C64 supercomputer, we have designed the Cyclops Datagram Protocol (CDP). CDP is a projection of the conventional network communication protocol (TCP/IP) to the modern C64 multithreaded architecture. It is a datagram-based, connection-oriented communication protocol that supports reliable and full-duplex data transfer. We have implemented the very popular BSD socket API in CDP. This provides a user-friendly programming environment for the C64 system or application programmers.

We have implemented the CDP protocol on the C64 thread virtual machine (TVM) [3]. The C64 thread virtual machine is a lightweight runtime system developed for the C64 chip. It provides the mechanism to map software threads directly onto the C64 hardware thread units. It also
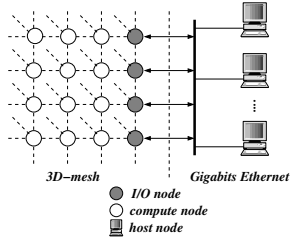
Figure 2. Cyclops-64 Supercomputer

provides a familiar and efficient programming interface for the C64 system programmers. Currently the C64 hardware is still under development, so the C64 thread virtual machine is running on the C64 FAST simulator [2].

We have explored a multithreaded methodology in the development of the CDP protocol. A fine-grain thread library called TiNy Thread (TNT) library [3] is used to implement the CDP protocol. The TNT thread library is part of the C64 Thread Virtual Machine and implements the C64 fine-grain thread model [3].

We have evaluated the performance of CDP through micro-benchmarking. From the experimental results, we have two observations: **(1)** The multithreaded methodology used in the implementation of CDP is very successful. It effectively exploits the massive thread-level parallelism provided on the C64 hardware and achieves good performance scalability. The speedup of a CDP test program can reach 82.55 after using 128 receiving threads. **(2)** As a communication protocol, CDP is efficient. The peak throughput of the user-level CDP (implemented by Pthread) is 884Mbps on the Gigabit Etherent.

In the next section, we will first introduce the necessary background of the C64 architecture. Then, we will formulate our problem and give a brief introduction to the solution.

2 Problem Formulation and Solution

A C64 chip has 80 "processors", which are connected to a 96-port crossbar network. See Figure 1. Each processor consists of two thread units, one floating point unit, and two SRAM memory banks (32KB each). A thread unit is a 64-bit, single issue, in-order RISC core operating at clock rate of 500MHz. The execution on the thread unit is not preemptable. A 32KB instruction cache is shared among five processors. The chip has no cache for data. Instead, a portion of the SRAM memory bank can be configured as scratchpad memory (SP), which is a fast temporary storage that can be used to exploit locality under software control. All of the remaining part of the SRAM form the global memory (GM) and is uniformly addressable from all thread units. The C64 chip does not support virtual memory.

The A-switch interface of the chip connects the C64 node to its six neighbors in the 3D-mesh network. In every CPU cycle, A-switch can transfer one double word (8 bytes) in one direction. The 3D-mesh may scale up to several ten thousands of nodes, which becomes a powerful parallel computing engine. The 3D-mesh computing engine is attached to the host system via Gigabit Ethernet and becomes the C64 supercomputer. See Figure 2. The whole C64 system is designed to provide computing power at Petaflops level. It is targeted at applications that are highly parallelizable and require enormous amount of computation.

Given the C64 multithreaded architecture and the C64 supercomputer organization, we are interested in two questions regarding the design and implementation of CDP:

- Is it possible to implement CDP in a way such that it can utilize the massive thread-level parallelism on the C64 hardware and achieve good performance scalability?

- Is the communication protocol we developed for the C64 architecture an efficient one?
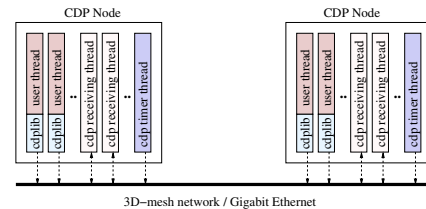

Figure 3. CDP Multithreaded Implementation

In order to answer these questions, we came up a multithreaded solution, shown in Figure 3. Briefly, the CDP program consists of a set of TNT threads [3]: the receiving threads, the timer thread, and the user threads. These threads cooperate with each other to implement the full functions of the CDP protocol. A fine-grain lock algorithm is proposed to improve the parallelism among these threads. Section 4 will give a detailed description on the CDP implementation.

The rest of the paper is organized as follows. Section 3 briefly introduces the CDP communication protocol. Section 4 discusses the multithreaded implementation of CDP. Section 5 presents the experimental results and analysis. Section 6 introduces some related works. Section 7 is our conclusion. We will talk a little about our future work in section 8.

3 CDP Protocol

CDP is inspired by TCP/IP, yet simpler and more compact. See Figure 5. Such a design is based on the consider-

2

ation that both the C64 architecture and the network topology of the C64 supercomputer are simple. We will briefly introduce the CDP protocol in this section and discuss the protocol implementation problems in the next section.

3.1 Overview

Figure 4 shows the position of CDP in the protocol stack. According to the OSI reference model, CDP corresponds to the *Transport* layer plus the *Network* layer. This implies that CDP should implement the main functions (or at least some) of these two layers that are specified in the OSI reference model. Here are the main features of CDP: (**1**) CDP is a datagram-based, connection-oriented communication protocol; (**2**) it is reliable and supports timeout retransmission; (**3**) it uses sliding-window based flow control mechanism to avoid network traffic congestion; (**4**) it provides a full-duplex service to the application layer; (**5**) it has implemented the very familiar BSD Socket programming interfaces for the CDP program developers.
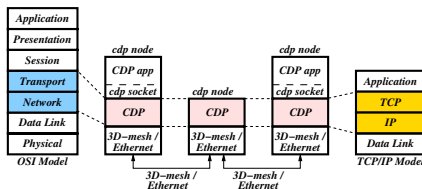

Figure 4. OSI Reference Model, TCP/IP Reference Model, and CDP Protocol Stack

3.2 Protocol Design

Figure 5 shows the CDP header format. The CDP header can be viewed as a merging of the IP header into the TCP header with certain customizations being applied.

In Figure 5, the *destination node* is used for addressing and routing. The 4-tuple <*destination node, destination port, source node, source port*> is used to identify a unique CDP connection, while the *"sequence number"* field identifies an individual CDP packet on a specific connection. CDP does not support selective or negative acknowledgments. So the receiver uses the *"acknowledgment number"* field to tell the other side that it has successfully received up through but not including the datagram specified by the *"acknowledgment number*. The *"flags"* field contains some control flags similar to TCP header.

We do not allow *fragment* and *de-fragment* in CDP. Furthermore, we also do not calculate checksum for the CDP datagram. This is because both underlying subnets are error-free.
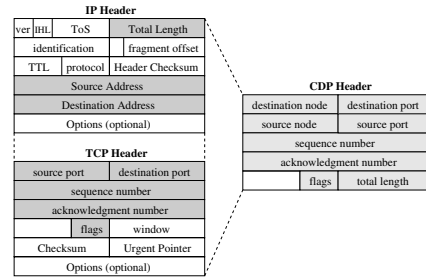

Figure 5. CDP Packet Header Format

As for the CDP connection, the finite state automata used to direct the connection state transition is shown in Figure 6. This finite state automata is similar to the one used in
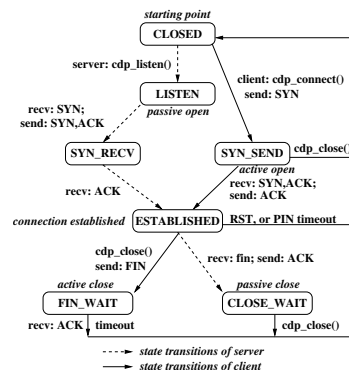

Figure 6. CDP State Transition Diagram

TCP/IP. The difference is that, instead of using the 4-way handshake protocol [11] to terminate a CDP connection, we employs a simplified 2-way handshake protocol. Because we do not want to support a "half-close" connection in CDP. This makes sense to most applications. When the user closes the connection at its side, usually it means that it does not have data to send out and does not want to receive data from the other side. So it is no problem to close the whole connection.

4 CDP Protocol Implementation

In this section, we introduce the multithreaded methodology used in the implementation of CDP protocol. To make the description easy to understand, we start from a brief introduction to the CDP programming interface. The CDP API is similar to the BSD socket API [12]. It consists of **socket**(), **bind**(), **listen**(), **connect**(), **accept**(), **send**(), **recv**(), and **close**(). The semantics of these functions are the same as their counterparts in the BSD socket API, except minor differences in the argument list. This implies that CDP supports the client/server programming model as well.

4.1 CDP Socket

The internals of CDP can be viewed as a collection of data objects (e.g. socket) and a set of TNT threads that operate on those data objects in parallel. See Figure 7. The most important data object is the CDP socket, or socket for short. Socket has two functions. First, it acts as the interface (through the file descriptor: *fd[]*) to the user. Second, it represents the CDP connection endpoint. All information about a CDP connection is maintained in the socket, and all operations on the connection are actually performed on the socket related. The sockets are linked into hash lists to improve the efficiency of socket searching. See Figure 7. The hash key is a function of the 3-tuple <*destination port, source node, source port*>. The hash function ensures that each hash list is evenly populated. Locks are attached to the socket and the hash list to guarantee mutually exclusive access.

4.2 CDP Threads

Figure 7 shows that there are three kinds of threads in a CDP program: user thread, receiving thread, and timer thread. These threads cooperate with each other to realize the full functions of the CDP protocol.
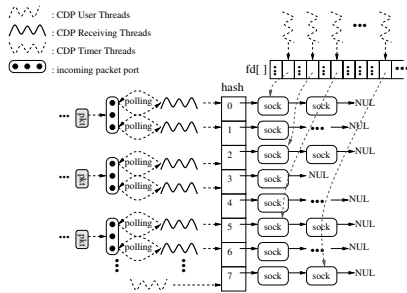

Figure 7. CDP Threads and Data Objects

The user threads are created by the user. They are not part of the CDP implementation. However, the user threads may call CDP API (*send(), recv(), bind(), listen()*, etc.) to access the internal CDP data objects, e.g. the socket. The user thread can establish a lot of CDP connections at runtime. But, at any moment, it can only work on one connection, i.e. operate one socket. The user thread accesses the socket through *file descriptor*, instead of traveling the hash lists to search for a socket. This follows the Unix convention. Sometimes, the user threads may insert a new socket into the hash list (by calling **accept()** or **connect()**), but they never delete sockets from the hash list.

The receiving threads are created by the C64 runtime system. They are TNT threads [3]. Therefore, their execution is not preemptable and can not be interrupted. The receiving threads always poll on the *"incoming packet port"*

for new incoming packets. See Figure 7. These *"incoming packet ports"* are the places where the underlying protocol handler put the incoming packets. All receiving threads perform the same routine: polling on a specific *port* for incoming packets; fetching a packet from the port if there is one available; searching the socket hash list and looking for the socket that needs to take this packet; processing the packet and the socket according to the operations specified by the CDP protocol. The packet is dropped or queued into the receiving buffer of the socket according to the result of the processing. The receiving threads neither insert sockets in the hash list, nor delete sockets from the hash list.

There is only one timer thread in the C64 runtime system. The timer thread is responsible for processing the synchronous events in the CDP program. A large number of these synchronous events are the timeout retransmission of CDP packets. Every one second, the timer thread is woke up from sleep by the hardware timer. It then traverses every hash list and visits every socket to handle the timeout events. If the timer thread founds that the current socket being visited is in *closed* state, or needs to be closed, it will remove the socket from the hash list. Timer thread will go to sleep again after it finishes visiting all the sockets in the program. Timer thread is the only thread that can remove socket from the hash list, but it never inserts new socket into it.

4.3 Parallelism

Generally, the performance of a network protocol is largely decided by the efficiency of the receiving side. This is easy to understand because it does not make much sense to send out more data if the receiver can not handle it. Therefore, the performance of CDP can be stated as: *"the number of CDP packets that can be processed per time unit by the receiving side"*. This can be characterized by the equation below:

$$P = \bar{N} \times t \times \rho(t) \tag{1}$$

In equation 1, $\bar{N}$ is the average number of packets can be processed by a single receiving thread per time unit, assuming that the underlying network link has infinite bandwidth. Its value is inverse proportional to the number of operations that need to be performed when processing one CDP packet. Actually, this is largely decided by the protocol design. t is the number of receiving threads used in the system. It is treated as a configurable system parameter. $\rho(t)$ is a factor that measures the parallelism in the program. $\rho(t)$ is a function of t. If t increases, $\rho(t)$ will decrease because the overhead of resource contention increases. The maximum value of $\rho(t)$ equals to 1 when t is 1. Generally, $\rho(t)$ is decided by the resource contention among the CDP threads. Higher contentions causes lower parallelism, which means

smaller value of $\rho(t)$. So, $\rho(t)$ is inverse proportional to the resource contention.

According to Figure 7 and the discussion above, two kinds of resources may limit the parallelism of a CDP program: the *incoming packet port* and the *socket hash list*.

The *incoming packet port* is implemented as a container data structure (list, queue, etc.). A lock is associated with each port to guarantee mutually exclusive access. The *incoming packet port* is the interface between the CDP receiving thread and the underlying device driver. Since the cost of copying a new CDP packet from the port is almost a constant, the only thing that may affect the performance scalability of CDP is the number of ports being used in the C64 runtime system. The experimental results show that one *incoming packet port* can support 16 receiving threads without harming the performance scalability too much. Section 5 has a very detailed discussion. Here we will concentrate on the *socket hash list*.

The access efficiency of the *socket hash list* has a great impact on the performance scalability of CDP. This is because all receiving threads need to traverse the *socket hash list* before they can do any operation on a socket. There are three kinds of operations performed on the hash list: socket insertion, socket deletion, and socket searching. The socket insertion operation happens when a connection is established and is only performed by the user threads. The socket deletion operation happens when a connection is closed and is only performed by the timer thread. These two kinds of operations are not as frequent as the socket searching operation, which happens every time when an incoming packet is received by a receiving thread. All these operations on the hash list need to be performed exclusively if they may cause data conflict.
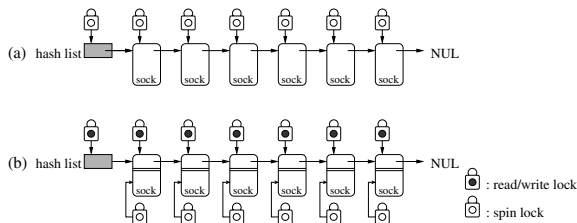

Figure 8. (a) Coarse-Grain Lock & (b) Fine-Grain Lock

The coarse-grain lock schema is shown in Figure 8(a). In this design, each socket in the hash list is associated with a spin lock to make sure that, at any moment, only one thread can operate on it. The integrity of the whole list is protected by a spin lock on the list head. No matter which operation (insertion, deletion, and searching) is performed, the thread first tries to grab the spin lock on the list head. If the thread failed to get the lock, it will busy wait. Otherwise, it will

hold it until the operation is finished, i.e. either the socket has been found or been inserted (removed) into(from) the list. After a thread has found the expected socket on the hash list, it also needs to grab the spin lock on the target socket before it can make any changes on that socket.

It is easy to see that, in the coarse-grain lock algorithm, the receiving threads are forced to traverse the hash list sequentially, even they may just search for different sockets. A more efficient solution is shown in Figure 8(b). The original spin lock is splitted into two: one is the read/write lock, which is used to protect the list pointer on the socket; the other one is the normal spin lock, which is used to protect the CDP connection related data fields. The lock on the list head is replaced with a read/write lock, to make sure that multiple threads can traverse the hash list simultaneously. When a thread wants to search a socket on the hash list, it does not need to lock the whole list. It only needs to *read_lock* the read/write locks on the current node being visited. When a thread wants to insert(delete) a socket into(from) the hash list, it needs to *write_lock* the read/write lock on the related list node to make sure that integrity of the list is maintained. For the socket insertion operation, the new socket is always inserted on the list head. Therefore, only the read/write lock on the list head needs to be *write_lock'ed*. For the socket deletion operation, two consecutive list nodes need to be *write_lock'ed*. They are the node to be deleted and the node previous to it. Because only timer thread can delete socket from the list, it is the only thread that tries to grab two locks at the same time. Therefore, deadlock will never happen. Although the socket insertion/deletion operation will force other threads that access the same socket to wait, it does not lock the whole linked-list. The threads that try to operate on other segment of the list can still proceed.

We did not consider using lock-free algorithms [13] [7] to implement the socket hash list. [13] uses extra *auxiliary nodes* in the linked-list to help implementing lock-free operations. This algorithm consumes more memory and makes the linked-list structure and operations more complicated than our algorithm. [7] depends on the hardware double-compare-and-swap atomic primitive which is not supported on the C64 architecture.

The fine-grain lock solution leverages the different linked-list access patterns of different types of CDP threads. The philosophy of this scheme is to make the common cases fast and keep the the whole design simple. In Section 5, we have compared the two design alternatives and shown that the fine-grain lock solution has much better performance scalability than the coarse-grain lock design.

5 Evaluation

We have designed two experiments to evaluate the CDP implementation. One is to investigate the performance scalability of CDP; the other is to assess the CDP throughput performance on the real hardware.

5.1 Performance Scalability

The first experiment is performed on the C64 FAST simulator. FAST is an execution-driven, binary-compatible simulator for the C64 multithreaded architecture. It can accurately model the functional behavior of each hardware component in the C64 chip. Although FAST is not cycle-accurate, it still estimates the hardware performance by modeling the instruction latencies and resource contentions at all levels of the C64 system. In the experiment, we measured the number of cycles that a CDP program need to process a specified number of CDP packets. We also measured the speedup obtained when using different number of CDP receiving threads.

The test case used in the experiment is a microbenchmark. At the beginning of the program, 128 connections are created. The number of connections is not fixed but fluctuates around 128 at runtime. This is to model the connection creation/termination events in the real world. Later, the specified number (1-128) of CDP receiving threads are spawned. These threads are not preemptable and can not be interrupted. Once running, the receiving threads poll on the *incoming packet ports* for new packets. An extra TNT thread is created to dynamically generate random CDP packets and feed them into the *incoming packet ports*. To make the experiment model easy to analyze, we assume that the underlying network link has infinite bandwidth and every thread read their own *packet port*. This implies that, when a receiving thread reads the port, a packet is always ready to be copied. There is no latency in between.

Figure 9 shows the performance scalability of the CDP protocol. We fed 256,000 packets into the program and spawned different number (1-128) of receiving threads to handle them. All packets have the same amount of payload: 1472 bytes. The figure shows both the execution time (cycle number) of the program and the speedup when using more than one receiving threads.

Figure 9 also shows the experimental results of two different design alternatives: the fine-grain lock and the coarse-grain lock. The figure tells that the performance of both versions scale up when we increase the number of CDP receiving threads from 1 to 128. However, the scalability of the fine-grain lock version is better than the coarse-grain lock version. For the program using fine-grain lock, the execution time reduced from 2162.8M cycles to 26.2 cycles after increasing the number of receiving threads from
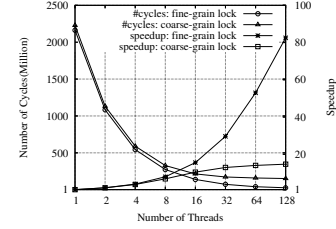

Figure 9. CDP Performance Scalability: Fine-Grain Lock vs. Coarse-Grain Lock

1 to 128. Thus, the speedup is 82.55. For the test program using coarse-grain lock, the execution time reduced from 2228.3M cycles to 151.8M cycles under the same condition. Its speedup is only 14.46, much less than the program using fine-grain lock. This means that the coarse-grain lock causes higher resources contention than fine-grain lock, as we have argued in section 4.
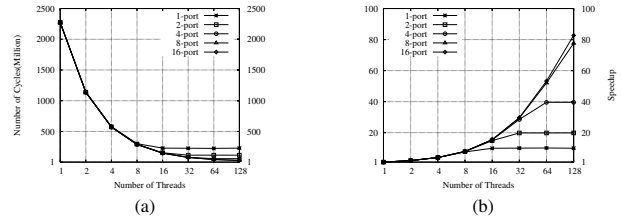

Figure 10. CDP Performance Scalability Under Different Number of Ports:(a) time curves (b) speedup curves

Figure 10 shows how CDP performance scalability is affected by the number of *incoming packet ports* used in CDP. We have measured the execution time (Figure 10(a)) and speedup (Figure 10(b)) of the test program when processing 256,000 packets (using 1 128 receiving threads) under different number of *incoming packet ports*. The results show that, when the number of ports is less than 8, the speedup will stop scaling before the number of receiving threads reaches 128. If the number of ports equals to 8, the speedup scales very well in the range of 1 to 128 receiving threads. To continue increasing the port number will not help improving the speedup. See the curves denoted as 16-port and 8-port in Figure 10(b).

We did not conduct the same experiment for test program with more than 128 receiving threads. Because there are only 160 thread units on the C64 chip. Some of them are reserved for other user/system tasks. Therefore, we believe 128 is a very aggressive upper bound for the number of receiving threads that we can use in a CDP program.

6

5.2 Throughput Performance

In order to evaluate the efficiency of CDP, we have designed an experiment which measures the throughput performance of a single-thread version of CDP on real machine. The throughput metric is important because CDP is supposed to be used in an environment where bulk data transfer is the majority network traffic. We have performed the same experiment for TCP and UDP, and found that the performance of CDP can compete with TCP and UDP, even CDP runs at user-level but TCP and UDP run in the Linux kernel.

Because the C64 hardware is still under development, we do not have the real C64 machine to run the CDP program. Meanwhile, the C64 simulator does not support full-system simulation [9] [10] [8]. It only simulate the architectural behavior of the C64 chip. Therefore, it is not possible to generate the true CDP performance number on the C64 simulator.

For this reason, we adapted the CDP TNT thread version to Pthread and ran CDP as a user-level program on Linux. We utilize the *packet socket* [12] interface (supported by all Linux distributions) to directly access the Ethernet device. Through this interface, we can encapsulate our proprietary CDP packet in the Ethernet frame and broadcast it to the Ethernet. Although the performance number obtained in this way is not 100% accurate, it still gives us enough insight into the CDP performance character.

The experiment was conducted on two compute nodes of a Penguin Performance Cluster, which is built by the Penguin Computing Inc.. Both of the nodes have the same hardware configuration (an AMD Opteron 200 processor, dual Broadcom BCM5721 Gigabit Ethernet cards, 4GB of ECC DDR SDRAM, with Linux kernel 2.4 installed). The test case is a client/server program. The client, using different protocols, tries its best to send as many datagrams (fixed size) to the server side as possible.
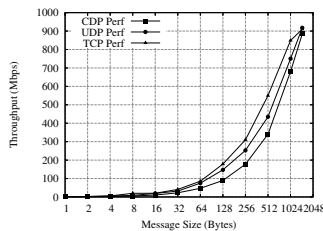

Figure 11. Throughput of CDP, UDP, and TCP under different message sizes: 1-1472 bytes

Figure 11 is the result of the experiment. The curves show the throughput of each protocol at different message sizes. The peak number of CDP throughput performance is 884Mbps on Gigabit Ethernet. For the other two protocols, the peak number of UDP is 920Mbps, and 927Mbps for TCP. All of these three protocols reach their peak performance number at message size 1472 bytes. We stopped at 1472 bytes because of the Ethernet MTU (1514 bytes). As we can see from the numbers, the peak throughput of CDP is a little bit smaller than TCP and UDP. This does not mean, however, that the design and implementation of CDP are poor. There are three reasons that can explain this result: **(1)** CDP has more interprocess context switches. The CDP test case needs at least 3 pthreads at runtime, while the test cases using TCP or UDP use only one Linux native process. Since the POSIX thread is implemented as a native process on Linux platform, there are more processes competing for processors in the CDP test case than in the TCP or UDP test case. **(2)** CDP has more memory-memory copies. In the CDP test case, user data need first be copied into the internal buffer of the CDP library, then be copied into Linux kernel space for further transfer. In the test cases using TCP or UDP, user data is directly copied into the kernel space. Compared with TCP or UDP, CDP test case needs two more memory copies in a send/receive session. **(3)** CDP has more kernel-mode/user-mode switches. In the CDP test cases, CDP library is running at user-level, while TCP and UDP are running at kernel-level. There are more kernel-mode/user-mode switches in the CDP test case than in the TCP or UDP test cases.

All of these are adverse factors that cause performance degradation in the CDP test case. However, these negative factors do not exist in the real C64 hardware platform. On the real C64 hardware, CDP protocol runs at kernel-level (as TNT threads) in the C64 thread virtual machine. There is no kernel-mode/user-mode switches, and no extra memory to memory copies either. Moreover, the TNT threads run on separate C64 hardware thread units and the execution of TNT threads are NOT preemptable. So, there is no competition for processors and no inter-process context switches. With these advantages from the real C64 platform, the performance of CDP will increase and may outperform TCP/IP. (currently, the peak performance of CDP is within the range of 95.4% of the TCP peak performance and 96.1% of the UDP peak performance).

In order to make an accurate comparison between CDP and TCP/IP & UDP/IP, we need to offset the negative effects caused by extra kernel-mode/user-mode switches and inter-process context switches in the CDP test case. We can achieve this in two ways: either add some "counter-balance code" in the TCP/UDP test cases, or directly implement CDP in Linux kernel. However, both methods are not quantitatively accurate. So, we do not have the motivation to make such a kind of comparison. After all, it is not our intention to design a new protocol to beat TCP/IP and replace it.

6 Related work

There are not many literatures concentrate on the implementation of a network protocol. [5] introduces the experience in running TCP/IP protocol stack on wireless sensor network. [6] is about the work on implementing TCP/IP on small embedded devices. These devices are usually 8-bit or 16-bit microcontrollors, not a multi-core chip like C64. [11] contains a thorough explanation of how TCP/IP protocols are implemented in the 4.4BSD operating system. [14] is a similar book that gives a comprehensive introduction to the TCP/IP implementation in the Linux kernel. However, all of these works are focused on implementing the functions and features of TCP/IP protocol that are documented in RFC. They seldom discuss the implementation methodology.

7 Conclusions

Based on the discussions on the experimental results, we have these conclusions: **(1)** Given a multithreaded architecture like C64, it is possible to develop a light-weight communication protocol for it such that the implementation of the protocol effectively leverages the massive thread-level parallelism provided by the hardware and thus obtains very good performance scalability. **(2)** The communication protocol we developed for C64 is efficient. The throughput performance of the single-thread (Pthread) version of CDP is 884Mbps on the Gigabit Ethernet, even it is running at the user-level on a Linux machine.

8 Future work

Currently, on the host side of the C64 supercomputer, CDP is implemented as a user-level protocol by using the Pthread library. Our next step is to move it to the kernel and incorperate the CDP module into the Linux protocol stack. This can enhance the CDP performance on the host side. In addition, on the C64 node, we will explore some efficient synchronization methods and try to exploit more thread-level parallelism.

Acknowledgments

References

[1] G. Almasi, C. Cascaval, J. Castanos, M. Denneau, D. Lieber, J. Moreira, and H. Jr. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 26–38. ACM SIGARCH, March 2003.

[2] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture. In *Workshop on Modeling, Benchmarking and Simulation (MoBS'05) of ISCA'05*, Madison, Wisconsin, June 2005.

[3] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Tiny threads: A thread virtual machine for the cyclops64 cellular architecture. In *IPDPS'05 - Workshop 14*, Washington, DC, USA, 2005.

[4] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Towards a Software Infrastructure for Cyclops-64 Cellular Architecture. In *HPCS 2006*, Labroda, Canada, June 2005.

[5] A. Dunkels. Full TCP/IP for 8 Bit Architectures. In *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, San Francisco, May 2003.

[6] A. Dunkels, T. Voigt, and J. Alonso. Making TCP/IP Viable for Wireless Sensor Networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004), work-in-progress session*, Berlin, Germany, Jan. 2004.

[7] M. Greenwald. Non-blocking synchronization and system design, phd thesis, stanford university technical report stan-cs-tr-99-1624, palo alto, ca, 8 1999., 1999.

[8] P. S. Magnusson. A design for efficient simulation of a multiprocessor. In *MASCOTS '93: Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, pages 69–78. Society for Computer Simulation, 1993.

[9] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb. 2002.

[10] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the simos machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[11] R. Stevens. *TCP/IP Illustrated, Volume 1&2*. Addison Wesley Press, 1994.

[12] R. Stevens, B. Fenner, and A. Rudoff. *Unix Network Programming: The Sockets Networking API, Volume 1*. Addison Wesley Press, 2004.

[13] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

[14] K. Wehrle, F. Pahlke, H. Ritter, D. Muller, and M. Bechler. *Linux Network Architecture*. Prentice Hall, 2004.