University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

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

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

CAPSL Technical Memo 074
February 14, 2007

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

A trend of emerging large-scale multi-core chip design is to employ multithreaded architectures - such as the IBM Cyclops-64 (C64) chip that integrates large number of hardware thread units, main memory banks and communication hardwares on a single chip. A cellular supercomputer is being developed based on a 3D connection of the C64 chips. This paper introduces our design, implementation, and evaluation of the Cyclops Datagram Protocol (CDP) for the IBM C64 multithreaded architecture and the C64 supercomputer system. CDP is inspired by the TCP/IP protocol. Its design is very simple and 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. It connects the C64 back-end to the front-end and forms a global uniform namespace for all nodes in the heterogeneous C64 system; **(2)** On a multithreaded architecture like C64, the CDP design and implementation effectively exploit the massive thread-level parallelism provided on the C64 hardware, achieving good performance scalability; **(3)** CDP is quite efficient. Its Pthread version can achieve around 90% channel capacity on the Gigabit Ethernet, even it is running at the user-level on a single processor machine; **(4)** Extensive application test cases are passed and no reliability problems have been reported.

Contents

List of Figures

List of Tables

1 Introduction

Cyclops-64 (C64) is a multithreaded architecture developed at the IBM T.J. Watson research center [6]. It is the latest version of the Cyclops cellular architecture that employs a unique multiprocessor-on-a-chip design [1]. It integrates a large number of thread execution units, main memory banks, and communication hardware on a single chip. See Figure 1. The C64

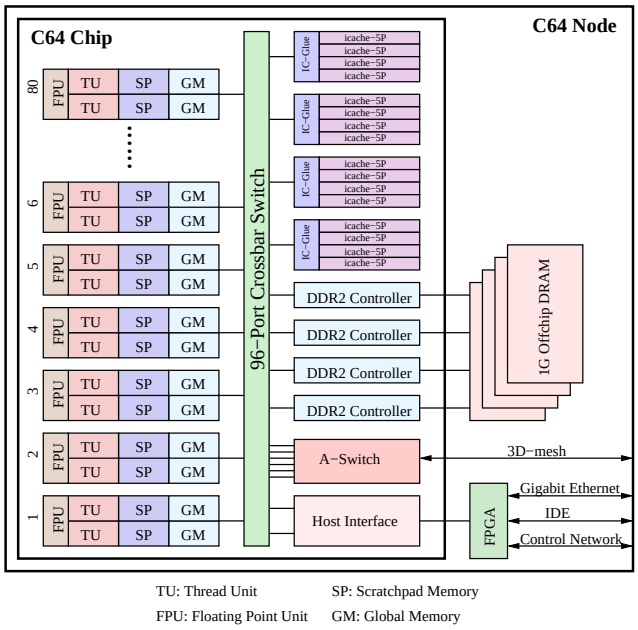

Figure 1: Cyclops-64 Node

chip plus the host control logics 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 petaflop computing power.

To interconnect the two different subnetworks in the C64 supercomputer, we 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. It supports reliable data transfer and provides a full-duplex service to the application layer. We have implemented the very popular BSD socket API in CDP. This provides a user-friendly programming environment for the C64 system/application programmers.

We have implemented the whole CDP protocol on the C64 thread virtual machine (the C64 TVM) [5]. The C64 thread virtual machine is a lightweight runtime system that resides inside the C64 chip. It provides a mechanism to directly map the software threads onto the C64 hardware thread units. It also provides a familiar and efficient programming interface for the C64 system programmers. Currently the C64 hardware is still under development, so the C64

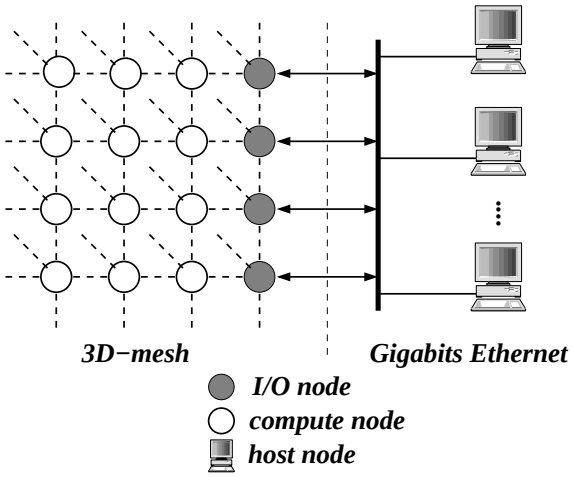

Figure 2: Cyclops-64 Supercomputer

thread virtual machine is running on the C64 FAST simulator.

We have explored a multithreaded methodology in the development of the CDP protocol. A fine-grain thread library called TiNy Thread (TNT) library [5] is used to implement the CDP protocol. The TNT thread library is part of the C64 Thread Virtual Machine. It implements the C64 fine-grain thread model [5].

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 benchmark that uses 128 receiving threads is 82.55. **(2)** As a communication protocol, CDP is efficient. A Pthread version of CDP achieves around 90% channel capacity on Gigabit Ethernet, even it is running at the user-level on a single processor Linux machine.

In the next section, we will give a problem formulation and briefly introduce the our solution.

2 Problem Formulation and Solution

As shown in Figure 1, a C64 chip has integrated 80 "processors", which are connected to a 96-port crossbar network. Each processor has two thread units, one floating point unit, and two SRAM memory banks, each 32KB. A thread unit is a 64-bit, single issue, in-order RISC processor 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. There is no data cache on the chip. Instead, a portion of each 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. There is no virtual memory subsystem on

the C64 chip.

The A-switch interface in 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 will form the powerful parallel compute engine of the C64 supercomputer. The C64 compute engine is attached to the host system through Gigabit Ethernet. See Figure 2. The whole C64 system is designed to provide petaflop computer performance. It is targeted at applications that are highly parallelizable and require enormous amount of computing power.

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

- Is it possible to implement CDP in a way such that it effectively utilizes the massive thread-level parallelism provided on the C64 hardware and achieve good performance scalability?
- Is the communication protocol we developed for the C64 architecture an efficient one?

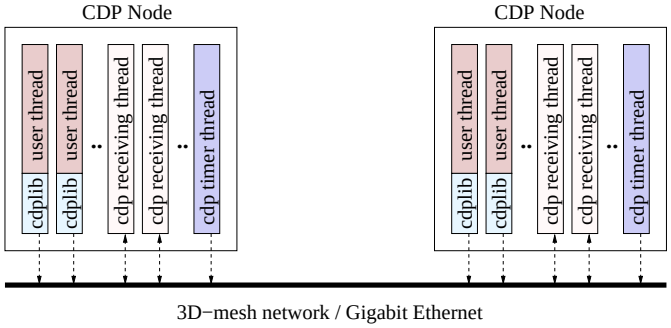

Figure 3: CDP Multithreaded Implementation

In order to answer these questions, we came up a multithreaded solution, shown in Figure 3. A CDP program at runtime consists of a set of TNT threads [5]: the CDP receiving threads, the CDP timer thread, and the CDP user threads. The receiving threads are responsible for handling the asynchronous events specified in the CDP protocol implementation, while the timer thread is to handle the synchronous events. The CDP code called by the user threads implements the semantics defined by the BSD socket API. These threads cooperate with each other to implement the full semantics and functions of the CDP protocol. A find-grain lock algorithm is proposed to ensure that operations on different CDP connection can be done in parallel. Section **??** will give a detailed description.

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. It is simpler and more compact. See Figure 5. Such a design is based on the consideration that both the C64 architecture and the network topology of the C64 supercomputer are simple. For the C64 chip, each thread unit is a single-issue RISC core. Its execution is not preemptable and there is no virtual memory subsystem. These features indicate that the C64 chip is not good at running complicate control intensive programs. Meanwhile, there are only two subnets in the C64 supercomputer system (Figure 2). One of them is reliable (the 3D-mesh), the other one is very stable (Gigabit Ethernet bit-error-rate is smaller than 10^{-10}). Given these properties, we are able to make some customizations to make the protocol simpler. This helps us to focus on studying our multithreaded implementation method.

It is not our intention to discuss the cutting-edge techniques for network protocol design in this paper. So, we will only briefly introduce the CDP protocol here and focus on protocol implementation 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. Below are the main features of CDP protocol:

- CDP is a datagram-based, connection-oriented communication protocol.
- CDP is a reliable communication protocol. It supports timeout retransmission on the CDP connection.
- CDP provides simple packet routing function.
- CDP uses sliding-window based flow control mechanism to avoid network traffic congestion.
- CDP provides a full-duplex service to the application layer.

The CDP library has implemented the very familiar BSD Socket programming interfaces for the CDP program developers.

3.2 Protocol Design

Figure 5 shows the CDP header format. Actually, the CDP header can be viewed as merging the IP header into the TCP header (or reverse) with some 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

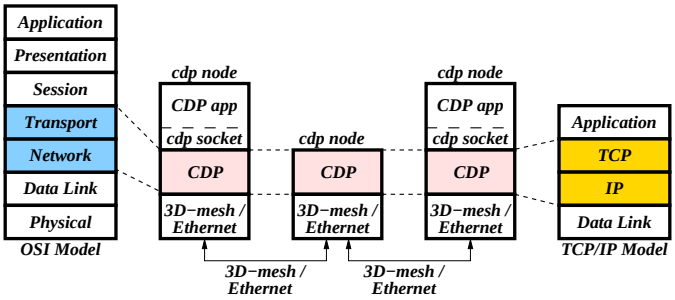

Figure 4: OSI Reference Model, TCP/IP Reference Model, and CDP Protocol Stack

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.

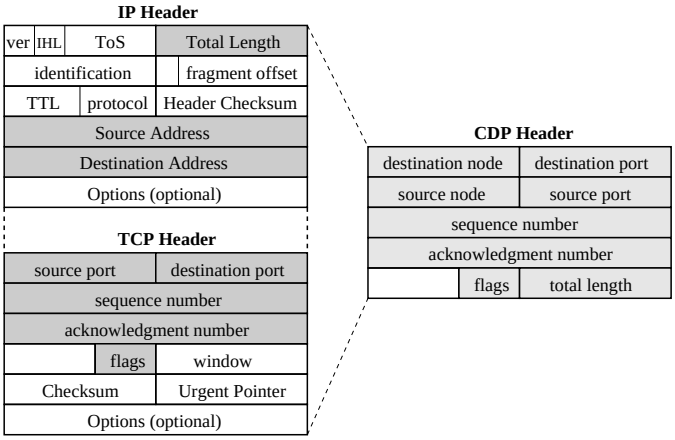

Figure 5: CDP Packet Header Format

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

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 TCP/IP. The difference is that, instead of using a 4-way handshake protocol [13] to terminate a connection, CDP uses a simplified 2-way handshake protocol. This is because we do not want to support a "half-close" CDP connection. This makes sense to most applications. When one end of the communication closes its connection, it always means that it does not have any data to send out and that it does not want to receive any data from the other side. So, there is no problem to close the whole connection. With this simplification, four states (*FIN_WAIT_2, CLOSING, TIME_WAIT, LAST_ACK*) are are removed from the CDP state transition automata.

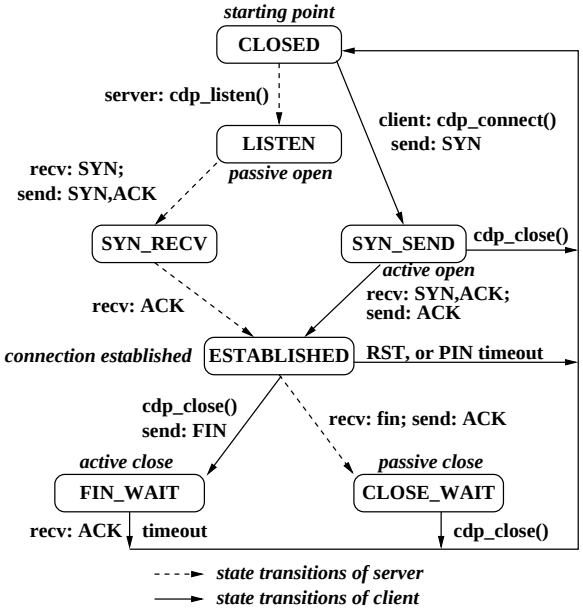

Figure 6: CDP State Transition Diagram

4 CDP Protocol Implementation

In this section, we will introduce the multithreaded methodology we used to implement the CDP communication protocol. (If not otherwise specified, the word "CDP" always refers to the implementation of CDP.) 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 them in parallel. Figure 7 shows the global picture of a CDP program at runtime.

4.1 Programming Interfaces

We have implemented the exact BSD socket API in CDP. They are **socket()**, **bind()**, **listen()**, **connect()**, **accept()**, **send()**, **recv()**, and **close()**. Except some minor differences in the argument list, the semantics of these functions are the same as in the BSD socket API [14]. Therefore, CDP supports the client/server programming model. Users can only access the *socket* data object through the *file descriptor*, which follows the Unix convention. See the *"fd[]"* array in Figure 7.

4.2 CDP Socket

The most important data structure in CDP is the CDP socket (or socket for short). Socket has two functions. On one hand, it is the interface (through the file descriptor: *fd[]*) to the user; on the other hand, it represents the CDP connection endpoint. All the important information about a CDP connection, like *address, state, receiving buffer, sliding-window*, etc, are

maintained in the socket. When the user wants to open a connection, a socket is created first. All the operations on the connection are actually performed on the corresponding socket. 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.

4.3 CDP Threads

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

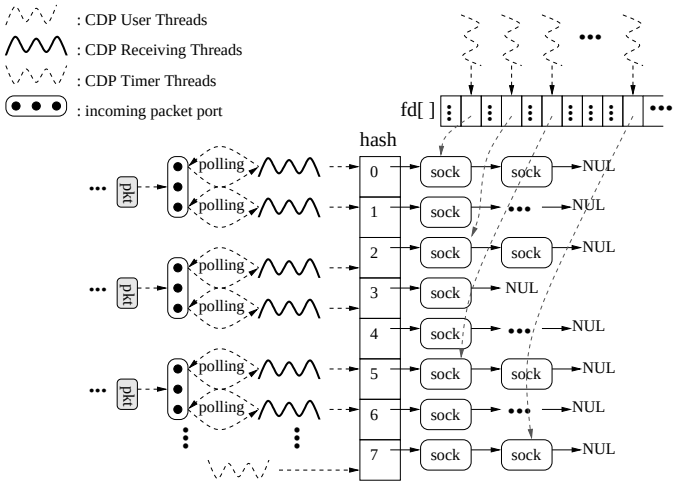

Figure 7: CDP Threads and Data Objects

The user threads are created by the user. There can be a lot of them. They are not part of the CDP implementation. But the user threads may call CDP API (*send(), recv(), bind(), listen()*, etc.) to access the internal CDP data objects, especially the socket. The user thread may establish a lot of CDP connections at runtime. However, in any transaction with the CDP module, it can only work on one connection, i.e. one socket. The user thread obtains the socket through a *file descriptor*, following the Unix convention. The user threads never travel the hash lists to search for a socket. 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 as TNT threads [5]. So the execution of receiving thread is not preemptable and can not be interrupted. They always poll on the *"incoming packet port"* for new packets. See Figure 7. These *"incoming packet ports"* are the places where the underlying protocol handler will put the incoming packets. All receiving threads are doing the same type of work: 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 need 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.

4.4 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 receive 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, there are two kinds of resources that may limit the parallelism of a CDP program: the *incoming packet port* and the *socket hash list*.

The *incoming packet port* can be implemented as a container data structure (list, queue, etc.). A lock is associated with each port to guarantee mutually exclusive access. The *incoming packet ports* are the interface between the CDP receiving threads and the underlying device drivers. Since the cost of copying a new CDP packet from the port is almost a constant, the only thing that may have great impact on the performance scalability of CDP is the number of ports 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 **??** has very detail discussion. Here we will focus on the *socket hash list*.

The access efficiency of the *socket hash list* has great impact to the performance scalability of CDP. This is because all threads need to access the *socket hash list* before they can do any operation on a socket. The only exception is that the user thread may access the socket directly through the *file descriptor*. 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.

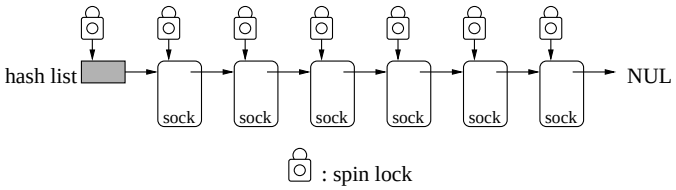

Figure 8: Hash List: Coarse-Grain Lock

A coarse-grain lock solution is shown in Figure 8. A spin lock is attached to every socket on the hash list. No matter what operation (insertion, deletion, and searching) is performed on the hash list, the thread first tries to obtain the spin lock associated with the list head. If the thread can not get the lock, it will busy wait; if the thread get the lock, it will hold it until the operation is finished. The spin lock on the socket is to make sure that no two threads operate on the same connection at the same time. So, after a thread finds the expected socket on the hash list, it will also try to obtain the spin lock on the target socket before it releases the lock on head node of the the hash list.

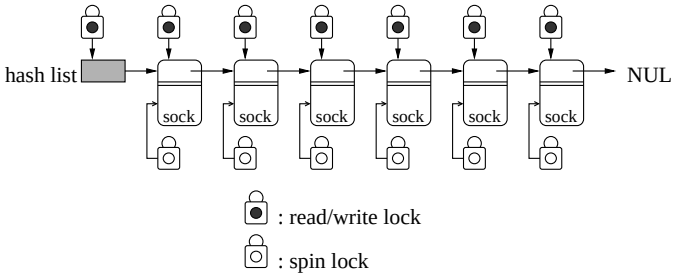

Figure 9: Hash List: Fine-Grain Lock

It can be easily figured out from the description that multiple receiving threads are forced to traverse the hash list sequentially, even they may search for different sockets. A more efficient solution is shown in Figure 9. The original spin lock is splitted into two: one is the read/write

lock that is only used to protect the list pointer recorded in the socket; the other is the spin lock used to protect the connection related data fields. The spin lock on the list head is also replaced with a read/write lock. This fine-grain lock scheme allows multiple receiving threads to traverse the hash list in parallel. When the receiving 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 attached to the current node being visited. When the user threads or timer thread wants to perform insert/delete operations on the hash list, they need to *write_lock* the read/write lock on the list node to make sure mutual exclusion is enforced. For the socket insertion operation, the new socket is always inserted on the list head. So, 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. Since only the timer thread can do socket deletion, it is the only thread that tries to grab two locks at the same time. Therefore, deadlock will never happen. Although a socket insertion or socket deletion operation will force other threads that access the same socket to wait, it does not lock the whole linked-list. The threads that work on other segment of the list can still proceed.

We did not consider using some lock-free algorithms [15] [9] to implement the socket hash list. [15] 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. [9] 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 CDP threads. The philosophy of this scheme is to make the common cases fast and keep the the whole design simple. In Section **??**, we compare the coarse-grain lock and the fine-grain lock scheme by experiments. The experimental results demonstrate that the fine-grain lock solution has much better performance scalability than the coarse-grain lock solution.

5 Evaluation

We have designed two experiments to evaluate our CDP implementation. The first experiment is to investigate the performance scalability of CDP; the second experiment is to assess the CDP throughput performance on the real hardware and compare it with that of the TCP/IP implementation in the Linux kernel.

5.1 Performance Scalability

One of the unique feature of CDP is its threaded implementation. The CDP program can spawn a specified number of CDP receiving threads (dynamically or statically) to handle the same number of CDP connections simultaneously. In this way, the CDP program obtains good performance scalability. The experimental results support our arguments.

Simulation

The 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 not only the functional behavior of each hardware component in a single C64 chip, but also the functions of multiple C64 chips that are connected in 3D-mesh. 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 this paper, our performance evaluation is restricted on a single C64 chip. This allows us only focus on a variety of design alternatives that may affect the performance of CDP protocol. The purpose of the simulation is not to analyze the microscopic runtime behaviors of CDP protocol. The goal is to evaluate the performance scalability of the threaded implementation of CDP on the C64 multithreaded architecture. According to this, we measured the number of cycles that a CDP program need to take to process a specified number of CDP packets and the speedup that 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. It fluctuates around 128 at runtime. This is to exactly model the connection creation/termination events in the real world. Later, the specified number (1-128) of CDP receiving threads are spawned. As we mentioned in the last section, these threads are programmed as TNT thread in the C64 virtual machine. So, the execution of receiving threads is not preemptable and may not be interrupted. Once started running, the receiving threads poll on the *incoming packet ports* for new packets. In addition to the receiving threads, an extra TNT thread is created to dynamically generate random CDP packets and feed them into the *incoming packet ports*.

To highlight the quality of CDP protocol, we assume infinite bandwidth of the underlying network links. Therefore, every time when a CDP receiving thread reads a port, there is always a packet ready to be copied. There is no latency in between.

Experimental Results

Figure 10 shows the performance scalability of the threaded implementation of CDP protocol. We fed 256,000 packets into the program and run different number (1-128) of receiving threads to handle them. All packets have the same amount of payload: 1472 bytes. The figure shows the time (cycle number) that the program used to process all the packets and the speedup obtained when using different number of receiving threads. We have presented in Figure 10 the experimental results of two different design alternatives: the fine-grain lock and the coarse-grain lock, which are described in detail in section **??**. The figure tells that the performance of both versions scale up well when the number of CDP receiving threads increase from 1 to 128. However, the scalability of the fine-grain lock version is better than the coarse-grain lock

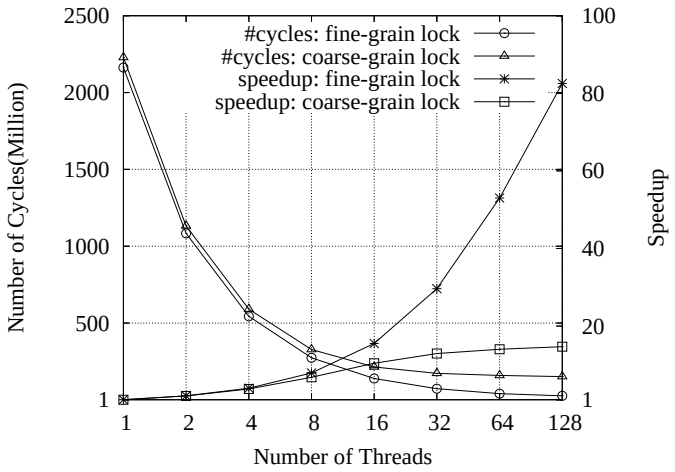

Figure 10: CDP Performance Scalability: Fine-Grain Lock vs. Coarse-Grain Lock

version.

For the test program using fine-grain lock, the number of cycles it takes to process 256,000 packets decreases from 2162.8M to 26.2M when the number of receiving threads increases from 1 to 128. Thus, the speedup is 82.55. While for the test program using coarse-grain lock, the number of cycles it takes to process the same number of packets decreases from 2228.3M to 151.8M when the number of receiving threads increases from 1 to 128. The speedup is only 14.46, much less than the program using fine-grain lock. This indicates that there are higher resources contention in the program using coarse-grain lock than using fine-grain lock, as we have argued in section **??**.

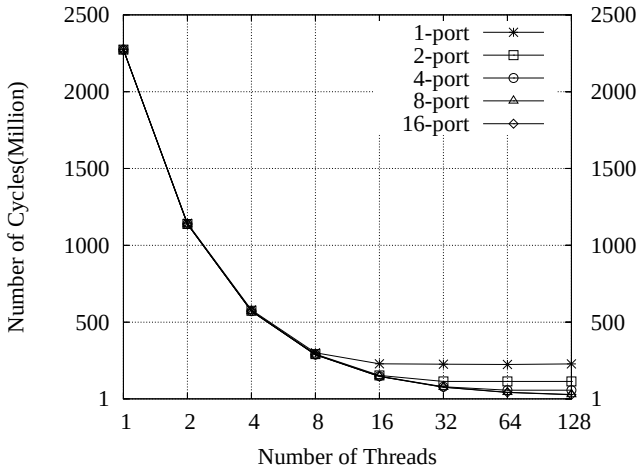

Figure 11: CDP Performance Scalability: Using Different Number of Ports (Time)

Figure 11 and Figure 12 show how CDP performance scalability is affected by the number of *incoming packet ports* used in CDP implementation. This is because multiple receiving threads may poll on a single port for incoming packets. Their accesses to this port are serialized through

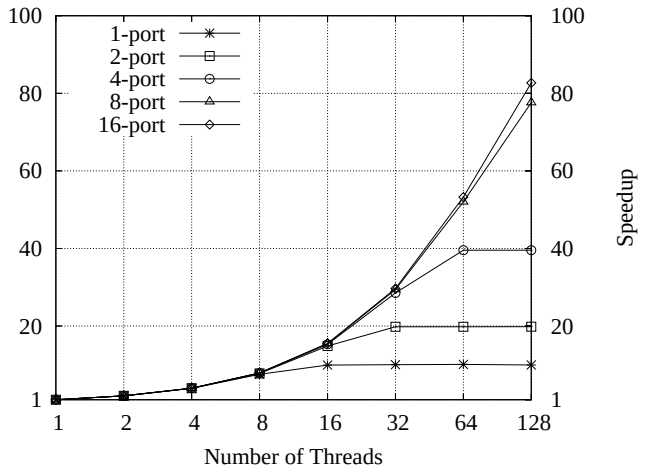

Figure 12: CDP Performance Scalability: Using Different Number of Ports (Speedup)

a lock associated with it. If the incoming packets flow into the C64 system in a speed that is faster than the receiving threads can tolerate, according to Little's law [2], the internal buffer of the underlying network devices will be exhausted soon. This may cause more packets to be dropt and thus hurt the the performance of CDP. The solution to this problem is to increase the number of *incoming packet ports* that can be used by the underlying network device drivers and the CDP protocol. The network device driver may put a new packet into a different and vacant port instead of waiting for an occupied one. Meanwhile, the receiving threads can also fetch packets from multiple ports in parallel. Therefore, in the same time unit, more CDP packets can be processed.

We measured the running time (Figure 11) and speedup (Figure 12) of the CDP test programs that processes 256,000 packets with 1 to 128 receiving threads under different configurations 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 12.

We have not done the same experiment for test cases with receiving threads more than 128. The reason is that there are only 160 thread units on a single C64 chip, and some of them need to be reserved for other user/system tasks. So, practically, we believe 128 is a reasonable upper bound for the number of receiving threads that we use in a CDP program.

5.2 Throughput

In order to evaluate the efficiency of CDP, we have designed experiment to measure the throughput performance of a single-thread CDP version 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. We compared the experimental results of the three protocols and found that the performance of CDP is comparable with TCP and UDP, even CDP runs as a user-mode program but TCP and UDP run in the Linux kernel.

Experimental Platform

Because the C64 hardware is still under development, we do not have the real silicon C64 machine to run the CDP program. Meanwhile, the C64 simulator does not support full-system simulation [11] [12] [10]. It only simulate the architectural behavior of the C64 chip. Therefore, it is not possible to generate the exact CDP performance number on the C64 simulator.

In this condition, we adapted the TNT thread implementation of CDP to POSIX thread and ran the CDP communication protocol as a user-level program on Linux. We use the *packet socket* [14] interface (supported by all kinds of Linux platforms) in the CDP library to access the Ethernet device directly. 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 made by the Penguin Computing Inc.. Both of the nodes have the same hardware configurations. To be more specific, an AMD Opteron 200 processor, dual Broadcom BCM5721 10/100/1000 Gigabit Ethernet cards, 4GB of ECC DDR SDRAM, with Linux kernel 2.4 installed. We developed the microbenchmarks to measure the throughput performance of CDP, TCP, and UDP. The CDP version of the test case is linked to the CDP communication protocol library. The other two versions are linked to *libc* to use the corresponding protocols. The microbenchmarks are client/server programs. The client tries its best to send fixed size datagrams to the server side (through different protocols) and see how many bytes can be transferred during a fixed amount of time.

Experimental Results

Figure 13 is the result of the experiment. The curves show the throughput of each protocol at different message sizes. The peak performance of CDP throughput is 884Mbps, which implements 88.4% channel capacity of the underlying Gigabit Ethernet. For the other two protocols, the maximum throughput of UDP is 920Mbps, and the maximum throughput of TCP is 927Mbps. All of these three protocols reach their peak performance number at message size 1472 bytes. We stopped at 1472 bytes because, under the limitation of Ethernet MTU (1514 bytes), this is the biggest user datagram that CDP can send. As we can tell from these numbers, the peak throughput of CDP is a little bit smaller than TCP and UDP. But this doesn't mean

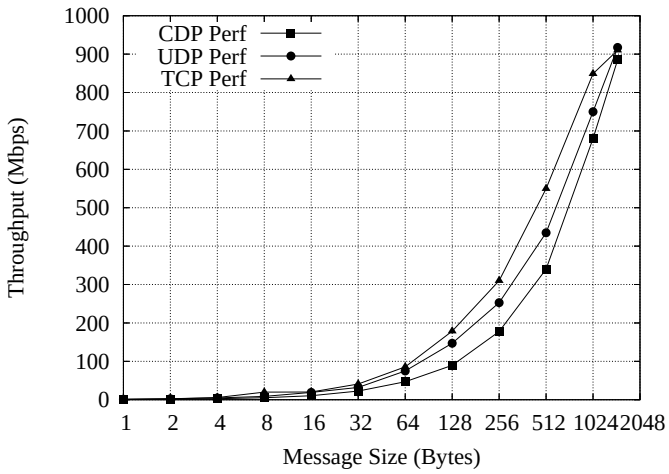

Figure 13: Throughput of CDP, UDP, and TCP under different message sizes: 1-1472 bytes

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. **(3)** 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 is running 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 adding some "counterbalance code" in the TCP/UDP test cases, or directly implementing CDP in Linux ker-

nel. 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. Our goal is to develop a simple and compact communication protocol for a special multithreaded hardware and explore a multithreaded methodology that can effectively exploit the massive thread-level parallelism on the hardware and achieve good performance scalability.

6 Related Work

There is some literature about the implementation of TCP/IP in a variety of computer systems. [7] introduces the experience in running TCP/IP protocol stack on wireless sensor network. [8] is about the work on implementing TCP/IP on small embedded devices. These devices are usually 8-bit or 16-bit microcontrollors. [13] contains a thorough explanation of how TCP/IP protocols are implemented in the 4.4BSD operating system. [16] 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. This paper explores the multithreaded method to implement a network communication protocol.

Here we make a brief comparison between the CDP implementation in the C64 thread virtual machine (or C64 TVM) and the TCP/IP implementation in Linux kernel. In the Linux kernel, interrupt is used to notify the arrival of a new packet [4], while in the C64 TVM, the CDP receiving threads poll on the *incoming CDP packet port* for new packets. Thus the CDP receiving thread responds instantly to the incoming packets. In the Linux kernel, the processing of an IP packet is splitted into two halves: the top-half is the urgent and fast interrupt handler [4] and the bottom-half is the slow and deferrable protocol handler [16]. But in the C64 TVM, the CDP receiving thread process a new packet without any stop until it is accepted or dropped. In the Linux kernel, the protocol handler is treated as a *softirq* [3] and is executed in the *ksoftirqd* kernel thread. The new packets need to wait until the *ksoftirqd* thread is scheduled to a CPU. Therefore, there is a longer latency between the arrival of a packet and its processing in the Linux kernel than in C64 TVM. Actually, the protocol handler is just one of the many tasks that need to be executed by the *ksoftirqd* kernel thread. In the C64 TVM, the CDP receiving thread is bound to a physical thread unit (no thread scheduling overhead) and is dedicated to processing incoming CDP packets. In addition, the number of CDP receiving thread is system parameter that can be changed. The C64 TVM can increase the number of receiving threads if the network traffic is heavy. This is not possible for the Linux kernel, in which the number of *ksoftirqd* kernel threads is a constant.

7 Conclusion

In the previous sections, we have reported our design, implementation and evaluation of CDP, a simple network communication protocol for the C64 multithreaded architecture which provides however all the necessary functionalities for real applications. We have also discussed our multithreaded methodology used to implement the CDP protocol. According to the analysis on the experimental results, we have these conclusions:

- Given a multithreaded architecture like C64 that has integrated a huge number of hardware thread units, we can develop a lightweight 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.
- The communication protocol we developed for the C64 multithreaded architecture is efficient. The performance of the single-thread version of CDP implemented by using pthread can achieve about 90% of the channel capacity on Gigabit Ethernet, even it is running at the user-level on a Linux machine.

8 Future Work

Another unique feature of the C64 multithreaded architecture is its segmented memory space. Each thread unit on the C64 chip has local scratchpad memory and on-chip SRAM. The on-chip memory has much shorter latency than the off-chip DRAM. See Figure 1. Meanwhile, all memory accesses to the off-chip DRAM or to the on-chip memory in the other processor need go through the 96-port crossbar switch, thus make the crossbar become a critical resource.

Our future work will try to improve the memory access efficiency of CDP by utilizing the non-uniform memory space feature of the C64 architecture. In addition, we also try to investigate how the resource contention for the crossbar switch influence the performance scalability of CDP. We will investigate the method to optimize the data layout of the critical data objects in the CDP implementation, and therefore decrease the contention for crossbar switch.

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] Dimitri P. Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall; 2nd edition, 1991.

[3] Daniel Plerre Bovet and Marco Cesati. *Understanding Linux Kernel (3 edition)*. O'Reilly Media, 2005.

[4] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers (3 edition)*. O'Reilly Media, 2005.

[5] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Tiny threads: A thread virtual machine for the cyclops64 cellular architecture. In *IPDPS'05 - Workshop 14*, Washington, DC, USA, 2005.

[6] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Towards a Software Infrastructure for Cyclops-64 Cellular Architecture. In *HPCS 2006*, Labroda, Canada, June 2005.

[7] Adam 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.

[8] Adam Dunkels, Thiemo Voigt, and Juan 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, January 2004.

[9] 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.

[10] Peter 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.

[11] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb. 2002.

[12] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the simos machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

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

[14] Richard Stevens, Bill Fenner, and Andrew Rudoff. *Unix Network Programming: The Sockets Networking API, Volume 1*. Addison Wesley Press, 2004.

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

[16] Klaus Wehrle, Frank Pahlke, Hartmut Ritter, Daniel Muller, and Marc Bechler. *Linux Network Architecture*. Prentice Hall, 2004.