Exploring mTCP based Single-Threaded and Multi-Threaded Web Server Design

A Thesis
Submitted in partial fulfillment of
the requirements for the degree of
Master of Technology
by

Pijush Chakraborty
(153050015)

Supervisors:
Prof. Purushottam Kulkarni
and
Prof. Sriram Srinivasan

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

June 2017

Approval sheet

This dissertation entitled **Exploring mTCP based Single-Threaded and Multi-Threaded Web Server Design** by **Pijush Chakraborty** is approved for the degree of Master of Technology in Computer Science and Engineering from IIT Bombay.

Examiners

[illegible] MYTHILI VUTUKURU

[illegible] VARSHA APTE

Supervisor

[illegible] PURUSHOTTAM KULKARNI

Chairman

[illegible]

Date: 28/6/2017

Place: IITB, MUMBAI

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Pijush Chakraborty

(Signature)

PIJUSH CHAKRABORTY

(Name of the student)

153050015

(Roll No.)

Date: 28/6/2017

Abstract

In client-server communications, every packet goes through a specialized processing via the TCP/IP Stack. Normally, the TCP/IP stack or the network stack is part of the operating system kernel and it does do the job well. But, the use of heavy data structures and system call overheads have made researchers wonder if the network stack can be improved. Moreover, with a single kernel TCP stack, the advantages of multiple hardware NIC queues are wasted. mTCP is a high-performing user space network stack which ensures that applications can take the advantage of having multiple NIC queues and have multiple TCP stacks(mTCP) on separate cores. The design of the mTCP architecture forces applications to be designed in a single threaded way so that it adheres to the no sharing model and processes packets of the respective queue. This project explores the possibility of a multi threaded traditional master-worker web server design using mTCP and how it can be implemented using a shared mTCP stack.

Contents

List of Figures

Chapter 1

Introduction

1.1 Need for User-space Networking

When it comes to client-server communication, it all boils down to having a client thread making requests and a server thread serving those requests. Each of these threads has to go through network stack so that the messages are correctly sent and received by the other end. The network stack or the TCP/IP stack processes message request and puts in headers and sends it to the other end where it again goes through the stack to find which process gets to see the message received.

Normally, the operating system kernel has its own network stack. So, whenever a packet arrives in the NIC, a specialized kernel driver(NIC driver) handles the packet and sends it to be served by the network stack. The TPC/IP stack of the kernel processes the raw packet and hands it over to the application thread via an eventpoll based mechanism.

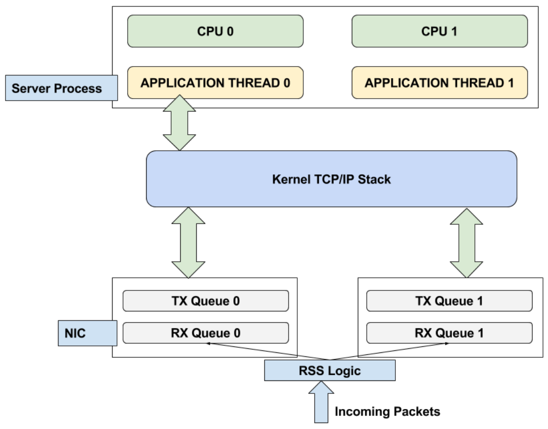

Figure 1.1: Kernel Network Stack

The kernel TCP/IP stack works fine but has a lot of overhead on heavy shared data-structures and system calls. Moreover, there is only one such stack to serve all the received packets. So, even if the hardware NIC supports 'N' hardware queues, there is just one stack to server them. This restricts the application to use only one listener queue. This surely is a bottleneck even if the application plans to have multiple threads serve multiple requests and this is certainly where a network stack in the user-space can be helpful. The following sub section on the problem definition is based on the multi-core network stack named mTCP stack.

1.2 Problem Definition

mTCP[2] network stack is an user-space network stack that serves as a high performing user-space TCP stack for client-server communication. The entire architecture as described in the next chapter is based on a no sharing architecture and two separate threads running on two separate cores will not have any shared data structure or events. With this design, mTCP has multiple threads working as network stacks pinned on separate cores.

The application using the network stack have to be single threaded to use the per core network stack in the following the way. Multiple instances of the single threaded application is pinned on separate cores which uses the mTCP threads as a network stack. The problem with such a design is that the TCP state is not shared between threads on two separate cores and thus master-worker server architectures cannot be easily ported to use mTCP as explained in the following chapters and requires some modifications to the mTCP stack.

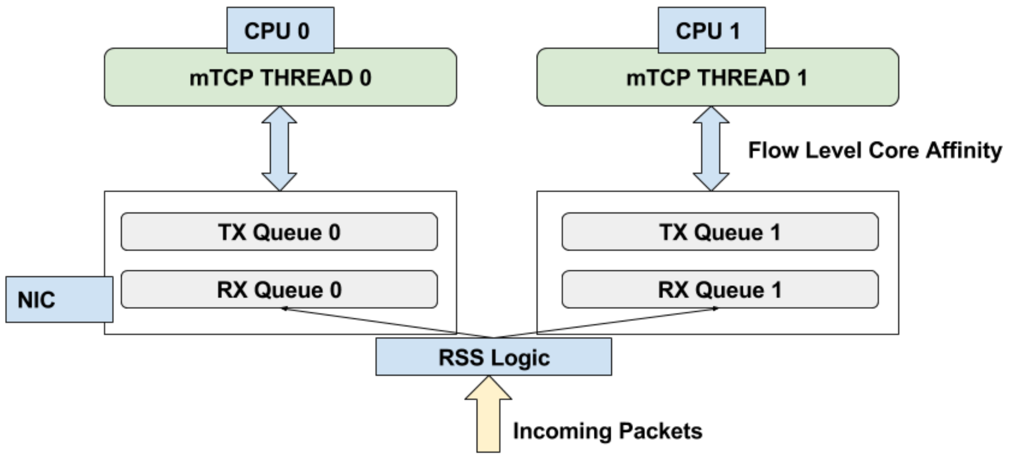

Figure 1.2: mTCP Design

Another goal that this project looks for is exploring lock free data structures such as RCU and RLU in the Linux Kernel. The main goals can be summarized in the following two points:

- Design and implement a way to port traditional multi-threaded master-worker servers to use mTCP and show its correctness.
- Compare performance of several single-threaded server processes to a single multi-threaded server process.
- Explore lock-free data structures such as RCU and RLU and find how significant it is in the Linux Kernel.

Chapter 2

Related Works

Userspace Networking is a new field and is currently being explored to analyze and effectively setup a usersace networking stack for high performance client-server communication. mTCP[2] is the first such high performing full fledged TCP/IP network stack that enables the clients to send data via mTCP stack and the servers to receive the same. This userspace receives raw packets using various fast I/O processing libraries such as DPDK[9]. The following chapters discusses the use of mTCP in detail.

A lot of work has been done in the field of lock free research. John D. Valois created the first CAS based lock free linked list implementation[10] that was improved[3] by Timothy L. Harris. The linked lists implemented by Harris was easy to use but lacked memory reclamation methods. The focus was then moved to design memory reclamation strategies such as Hazard Pointers[8] by M. Michael for such lock-less data structures. Combining lockless lists with memory reclamation strategies provided less performance than without the memory reclamation strategy but at the same time better performance than with lock primitives.

Read mostly research began to help improve the overall performance when the reads are relatively more than the writes. Paul E. McKenney invented Read-Copy-Update[7] with the goal of never blocking readers and allowing them to have a consistent view of the entire data structure. The mechanism provided memory reclamation after a wait-for-readers mechanism explained in the next chapter of this report. RCU proved to be much more efficient than other lock free mechanisms.

Chapter 3

Background

3.1 Introduction to mTCP

In the brief introduction in the previous section, we have seen that mTCP is an user-space high performing network stack which adheres to a no sharing model. The ’m’ in the mTCP is for multi-core or having multiple functional user-space network stacks pinned in separate cores. So, if the NIC has ’N’ hardware queues, there can be ’N’ mTCP threads running on ’N’ separate cores. mTCP uses DPDK for fast I/O handling and uses DPDK supported drivers to gain access to raw packets.

The entire design uses a strategy to pin a particular thread to a particular core so that it receives and sends packet from the NIC queue linked to that core. Moreover, it uses DPDK for all other network related I/O such as receiving and sending packets. The design of the mTCP architecture is shown in the following figure.

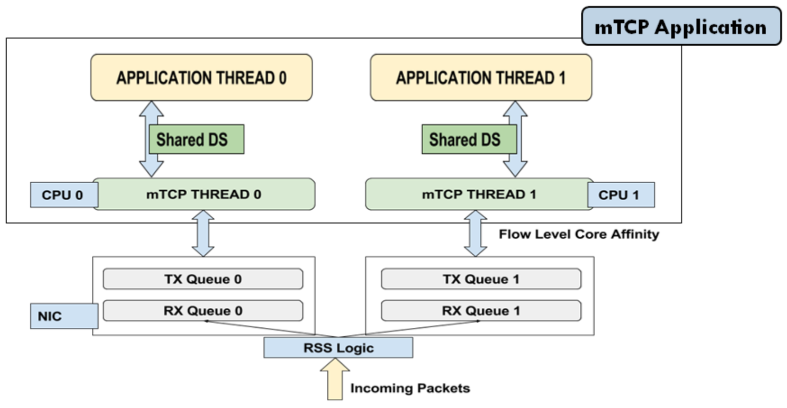

Figure 3.1: Introduction to mTCP Stack

As can be seen from the above figure, each mTCP thread sends and receives packets from the respective core. The application threads are designed to be single threaded to work well with the multi-core network stack so that they too can be pinned to each separate core and interact with the respective mTCP thread. The following section deals with the design of the mTCP thread and how the application interacts with it.

3.2 Design of mTCP based Applications

When it comes to applications using mTCP, it should be kept in mind that the design should be single threaded. mTCP provides a scalable eventpoll based mechanism to do the same. Moreover, as stated earlier there are no sharing data structures between threads running on separate cores and can be seen in the following figure. The below figure gives a description of how things flow from the mTCP thread to the application and how the application thread interacts with the mTCP thread.

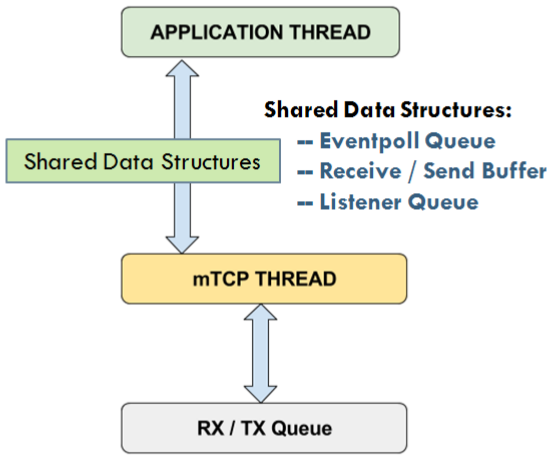

Figure 3.2: Fundamental Data Structures

As can be seen from the above figure, the application thread and the mTCP thread on a particular core share some data structures and related queues. Some of the important shared data structures are eventpoll queue and receiver/sender buffers. So, the only place where locks seem to be useful is between the application and the mTCP thread and only two threads compete for a lock. This use of no sharing model has helped to reduce the number of threads competing for a shared data structure and accordingly reduced the contention. The following section states the fundamentals of a few per core data structures mTCP applications uses.

3.3 Fundamental mTCP Data Structures

This section will discuss some of the most important per core data structures such as the eventpoll queue, the listener queue and the TCP state hash table to name a few. As discussed in the earlier section, when a packet arrives in the host NIC, the DPDK driver delivers it to the mTCP thread. The mTCP thread then proceeds as follows:

- The mTCP network stack searches for the packet stream from the TCP state hash table based on the source and destination information in the packet header. If the TCP state is not found, a new state or stream is created and inserted into the hash-table for further lookup.
- The mTCP network stack then starts processing the packet and if it is a connection establishment request, it simply inserts the stream into the per core listener queue[1] and also inserts the same event information into the eventpoll queue. The application which is waiting for the events then finally calls accept() and the stream is dequed from the listener queue and a wrapper socket is created. Moreover, this wrapper socket file descriptor is also a per core structure as the socket resides on that particular core only and cannot be shared between application threads.
- As described, the application thread waits for events to be served via an event-poll mechanism and as the mTCP thread processes the packets and according adds it to the network stack specific queue initially. It then gets hold of a lock and flushes all the events to a user specific events queue which is ideally what the application thread is waiting for.

As explained earlier, the data structures are per-core and application threads with the specific design are bound to use a single threaded design to fully make use the multi-core TCP stack. The eventpoll queue and the listener queue design can be further understood from the above figures. Moreover, as can be seen, the listener queue uses lock free queue to enqueue and dequeue streams as only one mTCP thread has access to it.

Chapter 4

mTCP based Application Design

4.1 Single Threaded Application Design

mTCP is designed to be per core and adheres to a no-sharing model and this bounds the applications to be designed in the same way. The simplest application thread design can be explained by the following figure.

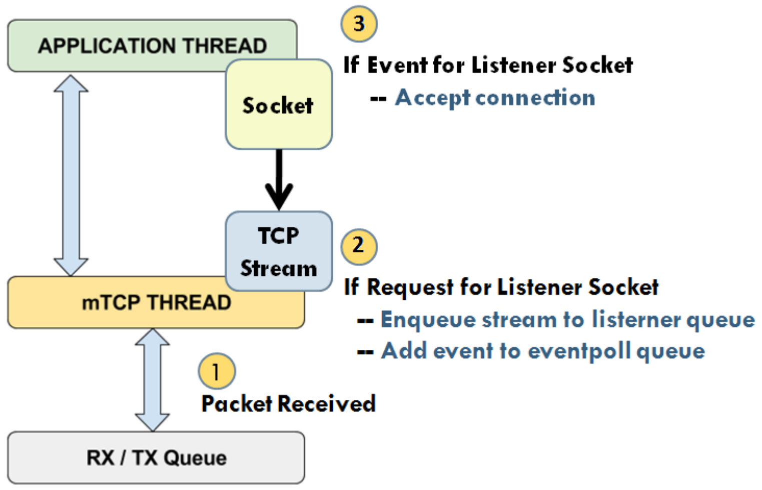
APPLICATION THREAD
Socket
3
If Event for Listener Socket
-- Accept connection
TCP Stream
2
If Request for Listener Socket
-- Enqueue stream to listerner queue
-- Add event to eventpoll queue
mTCP THREAD
1
Packet Received
RX / TX Queue

Figure 4.1: Single Threaded Design

As can be seen, each application thread is pinned to its particular core along with the mTCP thread. In the earlier chapter, we have seen how the application thread and the mTCP thread interacts with each other and how the entire flow of packets occur. This design enables a smooth lock-free way for packet processing and allows multiple mTCP threads to serve packets arriving in separate NIC hardware queues independently of the other.

Though, the design proves good for performance, traditional web servers having master-worker architectures are difficult to port to mTCP. Moreover, tradition web servers have blocking I/O in place and thus in addition to the shared design stated in this thesis we need a blocking I/O design to work with. In the following sections, we will look into how the present mTCP can be used to have such master-worker architectures in place and explore the possibilities of having another design to support the same.

4.2 Multi Threaded Application Design

As described in the earlier section, mTCP is designed for single threaded applications. For a master slave design, the master does the polling and allocates workers according to the need. With the present mTCP design, to have the master worker architecture in place, one can certainly have the worker threads to send all requests to the master and then the master decides on what action the worker should take. This design can be explained in the figure below.

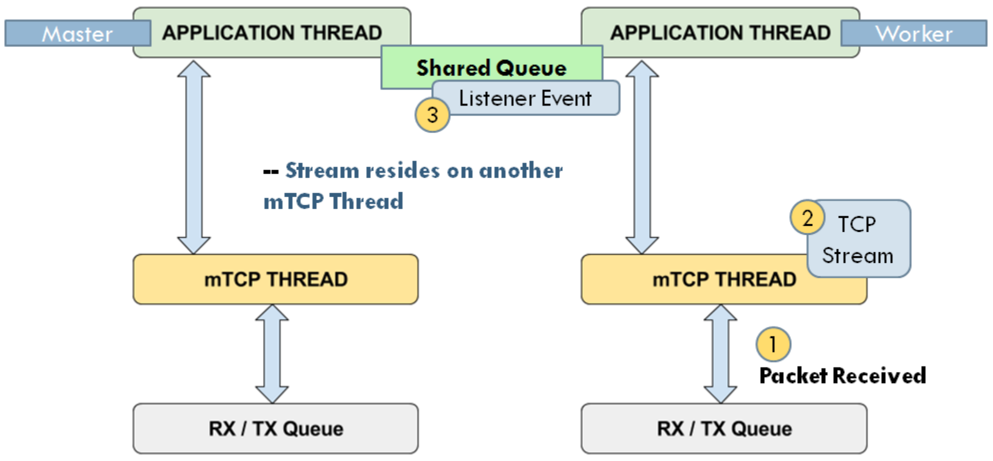

Figure 4.2: First Attempt to a Multi Threaded Design

The above design is clearly not an ideal way to do things as it would not allow the master to allocate a different worker for the job as the TCP state or the stream resides on the worker core. Moreover, the workers also need to go through the events on that core and thus needs to do polling too. Thus, we can never get any performance benefit of the multicore TCP stack using this design.

Another design can be to allow master thread and the worker threads to have access to the same mTCP thread. This seems a viable option, but will have only one network stack to process all incoming and outgoing packets. It boils down to be same as the kernel network stack with less heavy buffers and no system calls. So, this design cannot make use of the multiple NIC queues or the multi-core TCP stacks.

The following section deals with another design that takes use of shared data structures between two separate mTCP threads and tries to build a design that can make use of the multiple NIC queues and use multiple TCP stacks to work in parallel.

4.3 Shared mTCP stack to the rescue

From the discussions in the previous chapters, it can be said that the current design of the mTCP stack is not suitable for multi threaded master-worker application. As the name of the section suggests, we will explore a sharing based mechanism and explain how things can work for a master worker application. The entire changed design can be visualized from the following figure.

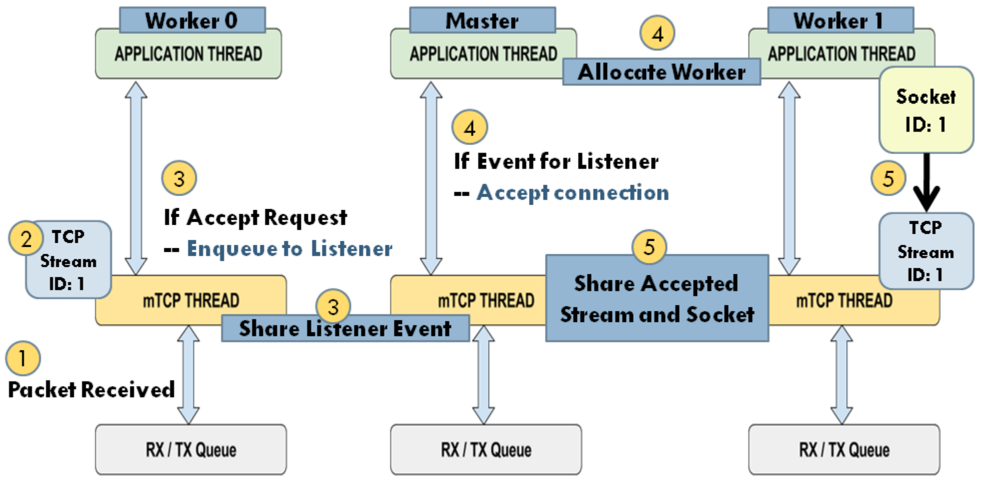

Figure 4.3: Master Controls Rverything

As can be seen from the above figure, the master is in charge for all poll related work and allocated workers to do the job. The design changes to the current mTCP stack can be summarized below:

- There is one listener queue[1] that accepts all new TCP connections and is shared among all mTCP threads. So, whenever a connection is to be established, the mTCP threads on separate cores simply enqueues into the shared listener queue.

- All events are flushed into a single eventpoll queue which belongs to the master thread. Once the master thread receives all such events it can select which worker to allocate for the job.
- The socket file descriptors are shared among multiple mTCP threads so that a worker can simple find the TCP state or stream simple by indexing a socket table.
- The current version of the changed mTCP stack allows the application to select which core will server the current socket file descriptor during connection establishment. All subsequent packets are processed by the worker core and events are again enqueued to the master eventpoll queue.

The shared data structures can be understood more clearly based on the following figure. As can be seen we have two set of data structures, one that is shared between two threads and one that is kept global. The use of the global data structure is explained earlier.

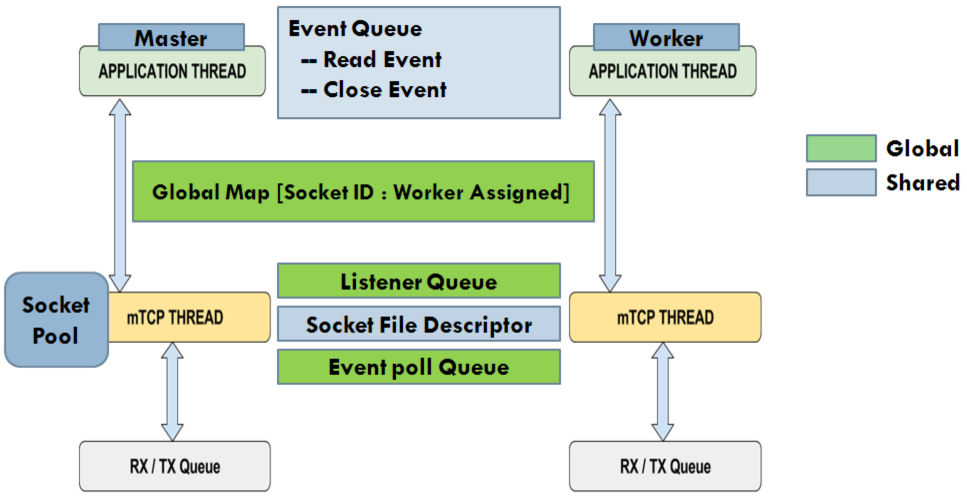

Figure 4.4: Multi Threaded Shared Data Structures

The figure below explains how a read event takes place. A packet arrives at one queue based on the RSS and accordingly the respective mTCP thread picks it up. If the state is established and the socket is allocated, the packet is given to the respective thread that is given the job to handle the socket.

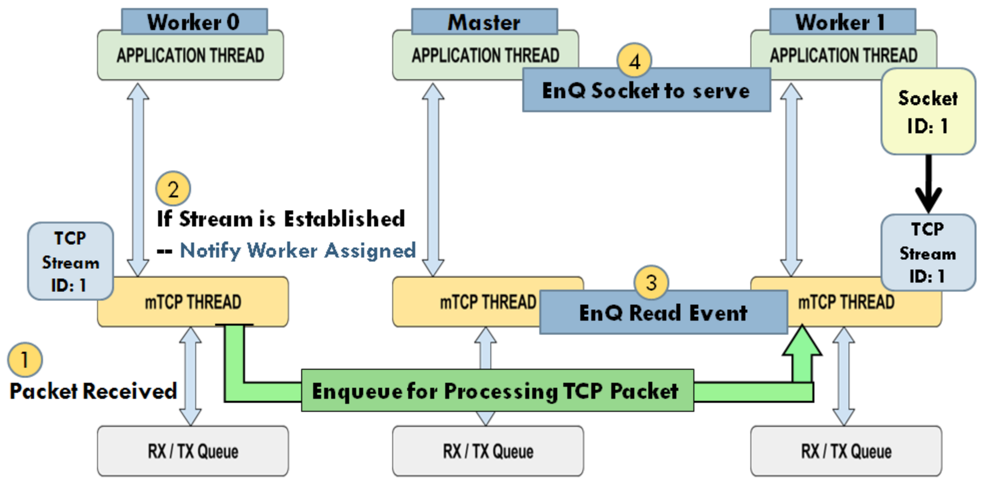

Figure 4.5: Arrival of Read Request from Client

The above design allows the master to take full control of the events from accepting connections to allocate read events. Moreover, the master alone is responsible for all polling whereas the workers are simply performing tasks assigned to them.

Another design to have a master worker architecture is to have only accept events controlled by the master and all the other events controlled by the worker threads alone. In this architecture, both the master and the workers will share the same listener queue and will both poll for events. Once a worker is allocated, it may work on its own based on the earlier design. The following figure shows how this design works compared to the first design.

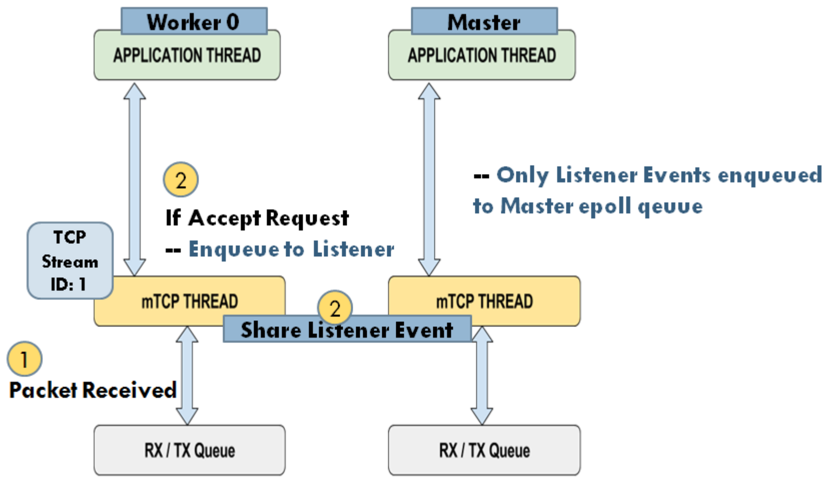

Figure 4.6: Multi Threaded Alternate Design

As can be seen from the above figure, in this design, the master and the worker both does polling and the master is only responsible for assigning which worker to allocate. Once the worker is allocated it functions normally as a single threaded eventpoll thread.

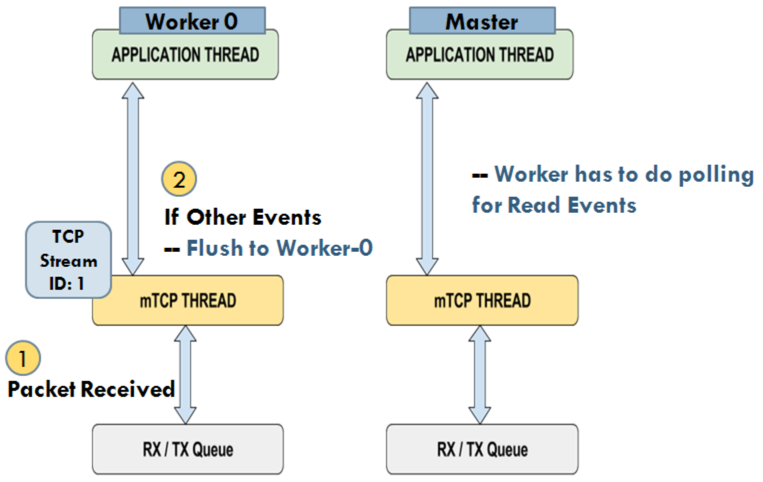

Figure 4.7: Read Event in New Design

So, if a packet comes and the state is established, it is enqueued into the appropriate worker's queue. If the state is not established, the stream is enqueued to the listeners queue and the master has to handle handle the event. In other cases, the workers can handle the read and write event easily.

Chapter 5

Performance Analysis

There are two experiments done to prove both correctness and compare performance of the web server designs. The experimental setup has both the client and server running on a 16 core Xeon Processor, 32GB Ram and 1GB Intel NIC. DPDK and other libraries are installed to make mTCP application work.

The first experiment is conducted to test the performance with varying file size requests from the client. The client and server details are given below:

- Client: 2000 open sockets on client side which sends requests of varying sizes for 20 seconds.
- Server: The server application is running the above machine with 7 cores assigned to it.

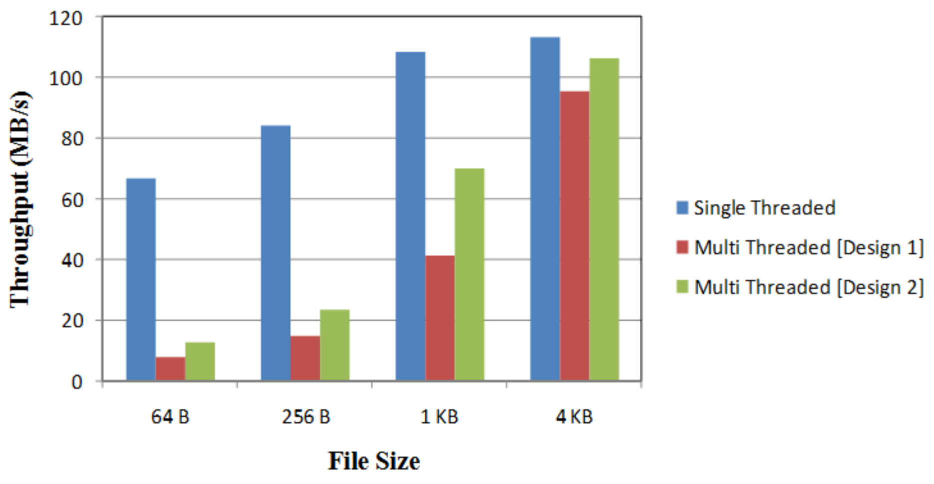

Figure 5.1: Performance based on varying file sizes

The second experiment is conducted to test the performance with varying cores on the server side. The client and server details are given below:

- Client: 2000 open sockets on client side which sends requests for 1KB file for 20 seconds.
- Server: The server application is running the above machine with varying cores.

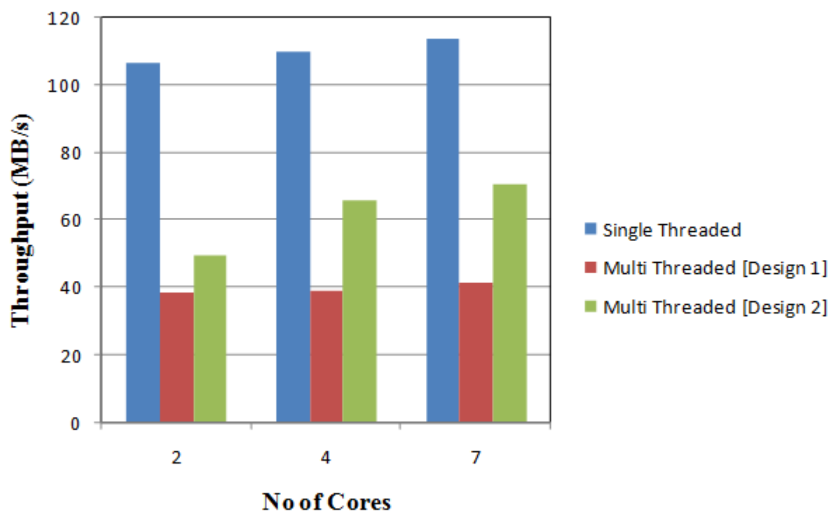

Figure 5.2: Performance based on varying cores

As can be seen from the above experiments, the first multi-threaded web server design which has the master controlling all events does not perform well compared to others. The master thread is surely the bottleneck in this case. For the second web server design, the performance is again not good compared to the single threaded design possibly due to use of shared inter core data structures.

Chapter 6

Exploring Lock Free Data Structures

6.1 Need for Lock Free Data Structures

In this new age of processors, multi-core processors and also multiprocessor(SMP) systems have made multi-threaded processes run efficiently and fast. Threads can now run in parallel on different processors allowing them to finish a task efficiently.

To allow processes to work, the treads must communicate with each other to ensure proper synchronization between them so that and the end result is consistent and as good as a single thread execution of the process. Multi-threaded programming offers a significant advantage in terms of efficiency but other methods for proper communication is required to ensure such consistency.

Threads in multi-threaded environment communicates using synchronization locks which allows one of the threads to access the critical section. The threads accessing the critical section blocks other parallel threads to safeguard the access to shared data structures or other memory regions. This may increase the overall completion time if all the threads tries to do some functional work on the same shared data structure. To reduce the overall completion time and help the threads gain maximum use of the available cores, a need for some lock free data structure arises.

A lock free data structure[10] is such that it doesn't use any mutex lock to block other parallel threads. It allows all threads to do concurrent updates or reads on a shared memory region such a shared data structure as shown in Figure 6.1. Though lock free data structures are common in use, there are certain problems such as the ABA problem which needs to be handled to ensure proper synchronization.

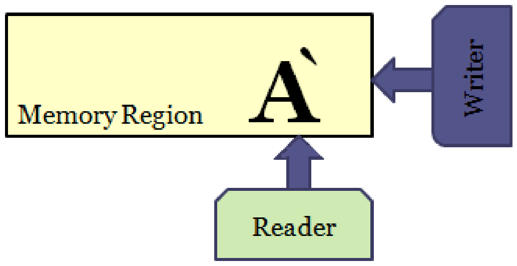

Figure 6.1: Readers and Writers concurrently accessing a memory region

The ABA problem mentioned earlier occurs in multi-threaded synchronization primitives deals with the problem when a particular reader thread reads a memory location while another writer thread concurrently deletes the original contents and adds something else. The reader in this case will still have access to the memory region, but with different content than when it first read it. There are solutions such as use of hazard pointers[pro haz] to solve the problem with certain overhead.

6.2 Optimizing Lock Free Data Structures

Over the last two decades a lot of work has been done on lock free data structures. Most of the work has led to algorithms that are simple to implement at the cost of memory leak[3]. A complete lock free data structure implementation with garbage collection proved to be costly in terms of performance but still provided better results than read-write locks.

Various research was done to work on data structures that was read mostly, i.e where the read count was much high than updates. The research carried out by IBM[pro] helped in developing Read-Copy-Update[7] that solves provides a lock free implementation and also an easy solution for the memory leak issue.

The RCU implementation also solves the ABA problem simply by maintaining multiple versions of the data structure and waiting for all old readers before freeing the original memory region. The performance of such an implementation proved to better than other lock free implementation targeting a generic data structure.

The main goal of this report is to understand certain behavioral aspects of such read mostly lock free data structures currently in use. The next section formally defines the problem statement of the thesis and the work that needs to be done for optimizing such lock free data structures.

6.3 Further Exploration

The main goal of this section is to explore based on the following set of questions:

- The first goal is to analyze and characterize the current RCU data structures in use and find any need of optimization. To find the need for optimization, certain experiments have been carried out that help us in the following goals.
- The second goal is to see if the writers can or should be helped. Measuring the current contention for updating an RCU protected data structure will tell us if the writer threads need any help or if the current synchronization primitives need any optimization. If the contention is much lower than expected we move on to the following goal.
- The third goal is to question if the list based semantics used currently is good for the readers or can they be replaced by certain array based semantics. The intuition here is that array based semantics provide better cache performance and also improves the overall performance.

Chapter 7

Lock Free Linked Lists

Looking inside the Linux Kernel, every non-blocking concurrent data structures is associated with the **Read-Copy-Update**[7] mechanism. RCU is a synchronization mechanism that solves the reader-writer problem with certain advantages in place. As we shall see later, RCU favors readers over writers and allows multiple readers to critical section with no worry about other parallel writers.

Another synchronization mechanism, Read-Log-Update has the same goal and also allow multiple writers easily and provide better throughput than Read-Copy-Update. The next few sections deals with both the synchronization mechanisms and the way they are used.

7.1 Introducing RCU

RCU provides an effective and efficient solution to the reader writer problem allowing multiple readers and writers to access the data structure. The mechanism never blocks the readers and is efficient for protecting read mostly data structures. The RCU mechanism mainly offers two main primitives for readers.

- **read-lock:** This primitive simply updates or increments the read count and doesn't actually provide any mutex operation. This primitive must be called by the reader before entering the critical section.

- **read-unlock:** This primitive is the one using which a reader should use to unlock the data structure and decrement the read count.

As for the RCU writers, RCU protected data structures must be updated so that readers are not harmed and must not be able to see any inconsistent data.

To make this work, Classical RCU approaches a simple mechanism of three steps:

- **Read:**
  Read or iterate over the data structure to find the memory region for updating it.
- **Copy:**
  Copy the memory region to a different location and then fiddle with it. This allows readers to work on the old memory region and cannot see any inconsistent data.
- **Update:**
  Update the old memory region atomically to the new memory region so that any new readers that comes along will see the new memory region and will again see consistent data.

In simple words, the writers copies the memory region, fiddles with it and finishes off by atomically replacing the old memory region. The main point to be noted is that, RCU maintains multiple versions of the same data structure at a particular point of time. The RCU writer update is done in a way to ensure readers are not blocked and it only provides one extra primitive, **synchronze-rcu**. This primitive is another important writer primitive that deals with removing old unused memory regions after all the old readers releases their lock on the data structure. This primitive will be discussed in detail in later sections.

7.1.1 RCU Phases

As described in the earlier section, the **RCU mechanism** maintains multiple versions of the data structure and ensures that new readers get to see the new updated data while old reader can still access the old data. To provide complete freedom to the reader, the **RCU Writer** works in three notable phases for updating a data structure as described below.

- **Removal Phase:**
  This phase is where the old memory region is read, copied and atomically updated so that new readers can see the updated version. The removal phase is simple and the completion of this phase makes multiple versions of the same data structure. Moreover, after this phase the readers are classified into old and new readers. The old readers are the one that started before the completion of the removal phase and the new readers are the threads that started accessing the data structure after the phase is completed. Figure 7.1 shows the removal phase and the classification between the readers.

- **Grace Period:**
  As soon as the removal phase is complete, new readers can see the updated version of the data structure. But, even now the old memory region may still be accessed by some old readers forcing the writer to wait for the old readers to exit before freeing the old memory region so that it can be reused. The grace period ends when the readers that started before the beginning of this phase(old readers) releases their lock. This phase is one of the important phase to prevent memory leak and the **Classical-RCU** approach to determine the period is discussed in later sections. This phase is dealt with the RCU primitive, **synchronize-rcu**.

- **Reclamation Phase:**
  This phase begins when the writer is sure that the old readers are no longer accessing the old memory region and it is safe to free the old memory region without any consistency problems. This phase simply frees the old memory region so that it can be reused later.

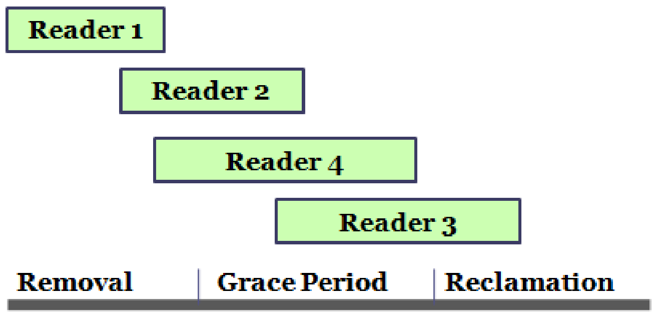

Figure 7.1: Phases in Read-Copy-Update Mechanism

As shown in Figure 7.1, the removal phase is where all the the functional work is carried out and after which the list is updated and is consistent for new readers to access. Now, since C doesn't have any garbage collection methodology, it all boils down to the writer for preventing memory leak. The second and third phase takes care of the garbage collection and frees the old memory region to be reused later.

7.1.2 Understanding the RCU Phases

The RCU mechanism has been described in earlier sections and the RCU writer phases have also been described. This section shows how the mechanism protects a simple linked list data structure. Suppose, the linked list in Figure 7.2, is protected by RCU and has two readers reading and iterating the list.

A writer thread has just started and wants to update the third node(node 'C'). The thread simply copies the node, updates it and atomically replaces the old node from the entire data structure as shown in Figure 7.3.
As shown in Figure 7.3, the simple atomic change of the next pointer of node 'B' atomically places the updated node 'C1' in the data structure. At this point there are two versions of the data structure starting from node 'A' and another version starting from node 'C'. Old readers still has access to node 'C' while new readers can access the original linked list having the updated version of the data(node 'C1'). This ends the removal phase and marks the start of the grace period.

The grace period waits for the old readers to release their locks on the memory region as shown in Figure 7.4. After the grace period ends, the writer can be sure the old memory region no longer has any readers referring to it and then it can proceed to memory reclamation phase.

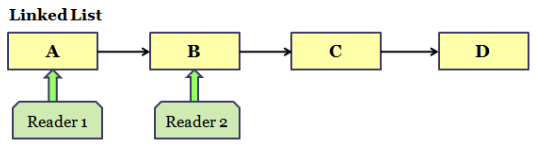

Figure 7.2: RCU Protected Linked List

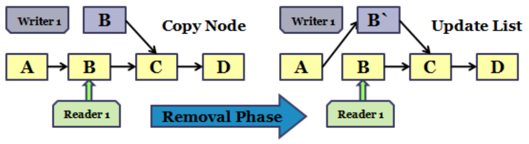

Figure 7.3: RCU Removal Phase

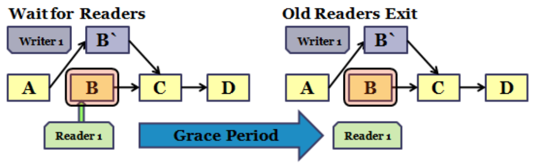

Figure 7.4: RCU Grace Period

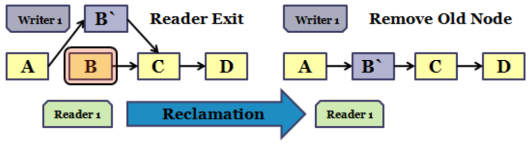

Figure 7.5: RCU Reclamation Phase

The final phase or the reclamation phase is shown in Figure 7.5, where the old memory region is removed to ensure there is no memory leak.

After the end of the last phase, the linked list remains consistent and the writer has finished the update. The old memory no longer can be accessed from the linked list and is no longer a part of the data structure. The next section describes how the grace period can be described and how the Classical-RCU determines it.

7.1.3 Issues with RCU Mechanism

The Read Copy Update Mechanism favors the readers allowing them to access the data structure without any worry about consistency. To provide this mechanism, normally writer synchronization is left to the developers using the RCU mechanism and these causes various problems as seen below:

RCU is not an efficient solution for data structures with more that one pointers such as a doubly linked list(Figure 7.6). The list shows that the removal phase is in process and a writer has copied Node-B and updated it. It now need to replace this updated node in the original linked list and for this the writer has to make two pointer assignments for **Node-A:next** and **Node-C:prev**.

Now, it must be noted that a single pointer assignment is atomic but assigning multiple pointers can not be atomic and leads to the state as shown in Figure 7.6.

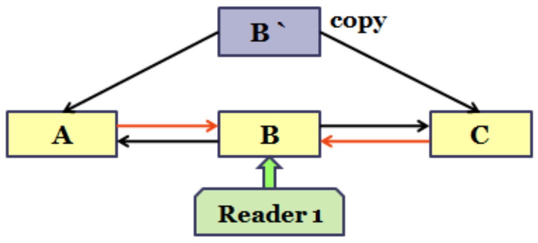

Figure 7.6: Issues with RCU

Here, To make the updated copy Node-B1 successfully replace the the old memory region Node-B, another pointer assignment must be done. At this point an old reader accessing Node-B has access to both the original and the updated version of the list which again provides inconsistency for the overall data structure.

The problems are solved by another new synchronization mechanism, Read-Log-Update(RLU) as described in the following section.

7.2 Introducing RLU

Read-Log-Update mechanism(RLU) [4] provides another efficient method for maintaining synchronization in read mostly data structures. Much like RCU, it never blocks the readers and at the same time provides better flexibility to writers. RLU allows multiple writers to make updates using a log based mechanism and can commit multiple changes to the data structure atomically. It overcomes the problem that RCU faces and provides an efficient version of synchronization as can be seen in later sections.

Another important advantage of RLU is that it can commit multiple objects atomically to provide readers accessing the data structure, consistent data over the entire data structure. As shown below in Figure 7.7, if RLU mechanism used the same concept of copying the memory region to update, then an old RLU reader will see **A-B-C-D** and a new RLU reader will see **A-B1-C1-D** as the protected list.

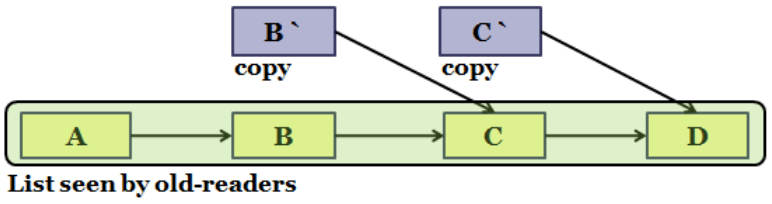

Figure 7.7: RLU Atomic List Update

This consistency is maintained using some lock based and clock mechanism described in later sections. Unlike RCU, where a reader may see any combination of the nodes, RLU presents a consistent data structure where a single operation allows multiple commits possible. Lets see the data structures used by RLU in order to achieve such consistency.

7.2.1 RLU Data Structure

RLU maintains various data structures in order to achieve the goals described in the earlier section. The data structures used by the RLU mechanism are given below:

- **Global Clock:**
  This is basically a software clock and it maintains a time-stamp denoting the current version of the data structure. Whenever a reader gets a lock on the region, it initially reads the current version of the time-stamp to later use it for iterating and finding the right memory region. The clock also denotes what each new thread should save as its version number.

- **Per Thread Data Structures:**
  RLU also has some per thread data structures as can be seen in Figure 7.8. The per thread data structures are useful for the thread to know how to use the data structure and accordingly make decisions.

The per thread data structure is quite important for maintaining synchronization among threads. Each thread maintains a per **thread clock** which it initializes from the global clock. This thread clock validates the time when the thread has started reading and according make decisions in the long run. The second per thread data

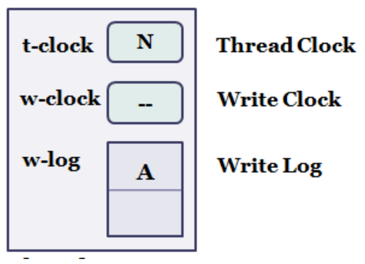

Figure 7.8: RLU Per Thread Data Structure

is the **write clock** which is used by the thread when its updates a memory region. The write clock signifies the time stamp when the writer has issued a commit to signify that it has completed the update. Each thread also maintains a **write-log** which is simply holds a copy of the memory region to be updated much lick what RCU does. The log maintains copies of all memory region it wants to update.

Along with the above data structures, each memory region has an associated header that stores information related to the writer thread. The important information includes which writer-thread currently has a lock on the region and a pointer to the updated copy of the memory region in the log.

All the data structures described above is used by the threads to maintain synchronization and will be described in later sections. Another important point to note is that RLU objects are protected by fine grained locking which allows the multiple writers to work on the same data structure with fine grained locks on each node to allow multiple writers update the same shared data structure. The next section describes how threads use the data structures to maintain synchronization and consistency.

7.2.2 RLU Readers

RLU Readers are not blocked by other threads and are provided a consistent view of the entire data structure. The reader can be summarized as below:

- The reader when started initializes its per thread clock with the global clock. This is to determine when the thread began reading.
- It then proceeds to iterate the data structure such as the linked list. If it finds a memory region currently locked by ta writer thread, it checks if **Reader.t-clock>=Writer.w-clock** and if so reads the data from the writer log. The comparison basically determines if the reader thread started before the writer committed or after the commit.

So, as with RCU, old RLU readers too get to access the old memory region while new readers gets to see the new memory region. RLU Writers work in a different way as shown below.

7.2.3 RLU Writers:

RLU Writers work in a different way. The writer thread can also be summarized as shown below:

- It initially starts by initializing its per thread t-clock with the global clock and its w-clock to infinity(possibly a large value) and then iterates the data structure to the memory region which it wants to update.
- The writer updates the memory region header and updates the lock field to its own thread-id. It then copies the data to the write-log and updates it. The writer commits by setting the w-clock=g-clock+1 and the global clock(g-clock) is also incremented by 1. The commit clock update is done to ensure that any new readers that tries to read a locked object can read the updated data in the log as any new reader will have its per thread t-clock more than the locking thread's w-clock. Thus the writer maintains consistency and as RCU writers relies on atomic pointer assignments, RLU writers relies on atomic clock updates.

After the writer thread commits, the write-log consists of the updated data which must now be copied back to the original memory region. The writer now has to wait for all the readers to exit and this phase is similar to the RCU Grace Period. Moreover, the RLU Grace Period can be determined the same way as the RCU Grace Period. RLU Writers may defer the write back or delay the copying of the updated data to the original memory. It may be noted that as in **Hazard Pointers** [8], RLU uses the same strategy to use a reference to a log in order to defer its removal.

Chapter 8

Comparing two Lock Free Mechanisms

8.1 An empirical Comparison of Read-Copy-Update and Read-Log-Update Mechanism

While Read-Copy-Update is be used heavily in the Linux Kernel, it should be noted that Read-Log-Update provides some better results in terms of overall performance as claimed by the creators of Read-Log-Update mechanism.

This section of the report tries to answer the a few set of questions which will enable us to understand if the current lock-free Read-Copy-Update implementation used in the Linux Kernel can be or rather should be improved. The experiments are done on RCU protected linked lists and hash lists as these are the main data structures protected by RCU in the Linux Kernel.

The following is the set of areas which needs attention in this section

- As per the claims of the authors of Read-Log-Update, it performs better than Read-Copy-Update due to the use of fine-grained locks for writers to use. In this part we verify the claim that Read-Log-Update provide better performance than Read-Copy-Update mechanism when used with shared data structures.

- Moreover, the Read-Log-Update mechanism requires copying the contents from the original location to the log and again back to the original memory location. This overhead in having a second requirement for copying memory was done to avoid ABA problem. This second part compares the performance of both the mechanisms with varying node sizes.

8.1.1 Comparing Performance of Read-Log-Update and Read-Copy-Update

As mentioned earlier, this part will try to verify the claims of Read-Log-Update and answer a few other questions. The experiments were conducted on a system with Xeon 16-core blade server supporting 16 hardware threads. Each of the following experiments tries to compare the number of operations done with varying no of threads. The maximum number of hardware threads is fixed to be 16 for the system we are working on.

The experiment uses a synthetic benchmark used by the RLU authors[4] that inserts,deletes and reads from a list based data structure randomly. The benchmark can operate on any shared data structure and can be fitted with a new implementation of lock free mechanism with varying parameters.

The operations noted in the results are done with varying number of threads and varying update rate. The following are the results with various update rate for a simple linked-list with 10000 initial nodes and fixed node size.

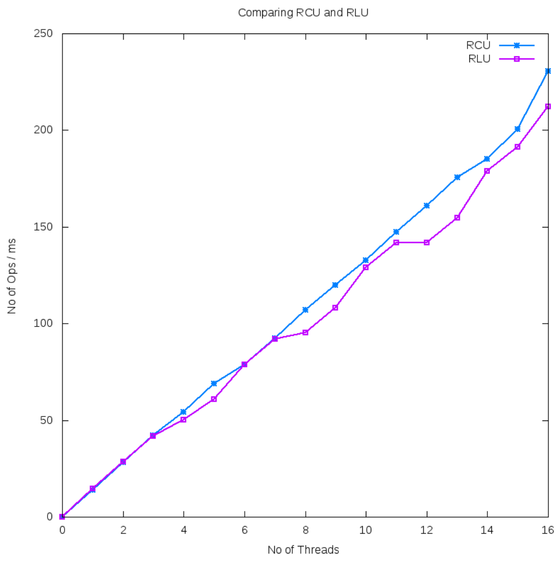

Figure 8.1: Throughput for RCU and RLU Linked lists with no updates

As can be seen from the above plots, RLU has better throughput for linked lists. The claim done by the creator of RLU mechanism is thus verified for linked list.

The same experiment was again conducted with an hash-list with 20 buckets and

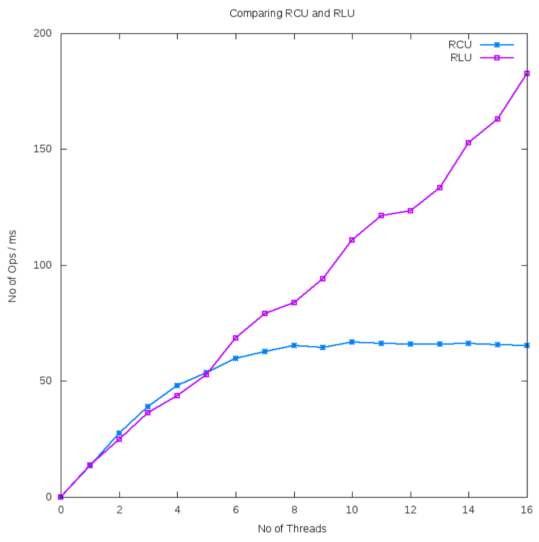

Figure 8.2: Throughput for RCU and RLU Linked lists with 20% updates

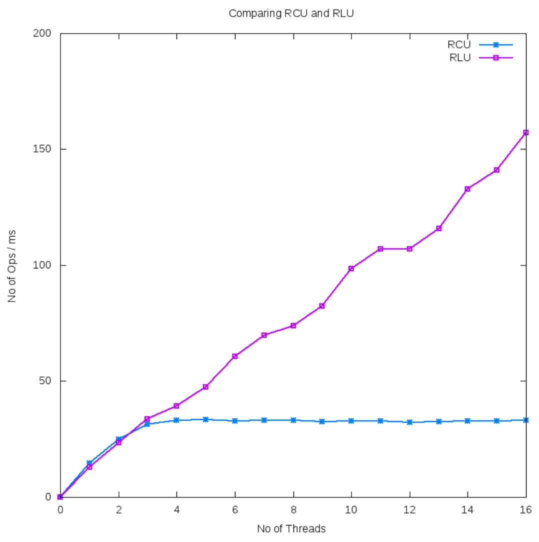

Figure 8.3: Throughput for RCU and RLU Linked lists with 40% updates

10000 initial insertions. Hash-lists are organized as a linked-list with each key having belonging to a separate bucket.

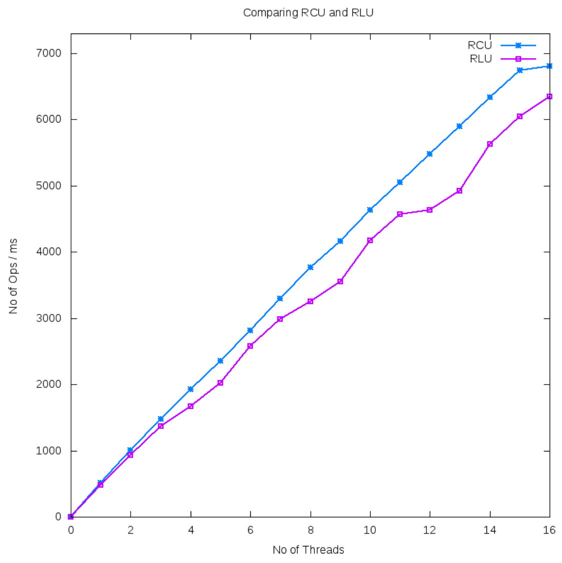

Figure 8.4: Throughput for RCU and RLU Hash lists with no updates

As can be seen from the above plots, RCU has better throughput for hash-lists and is also shown by a different experiment[6] by Paul E. McKenney. This can be

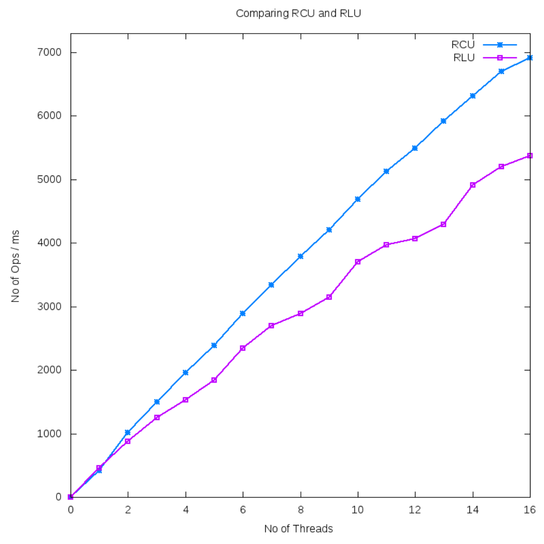

Figure 8.5: Throughput for RCU and RLU Hash lists with 20% updates

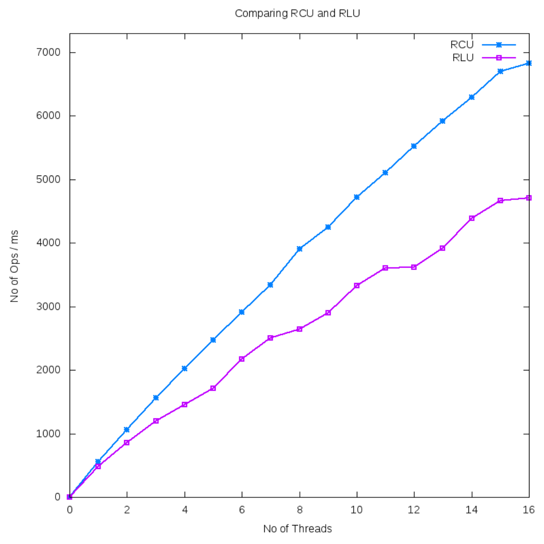

Figure 8.6: Throughput for RCU and RLU Hash lists with 40% updates

because RCU allows multiple writers to easily work on separate buckets where as RLU allows multiple writers with certain overheads.

Thus, it can be said that use of RCU for protecting hash-list is still favorable to use. The next part looks into the influence of node size on the overall performance of protecting linked lists.

8.1.2 Change in Throughput with Node Size

This part tries to find out if the node size of linked lists has any effect on the performance of Read-Copy-Update or Read-Log-Update mechanisms. RLU mechanism requires copying the contents twice from original memory location to the writer log and again back to the original location to avoid the ABA problem. The cost over this overhead with increasing node size is thus a good area to explore.

The experiments were conducted on a system with Xeon 16-core blade server supporting 16 hardware threads. Each of the experiments tries to compare the number of operations done with varying node sizes and fixed number of threads. The experiment uses the same benchmark as in the above part.

The following are the results of finding the relative performance of the two lock free mechanisms with increasing node size and with varying update rate for a simple linked list with 10000 initial nodes and fixed number of threads.

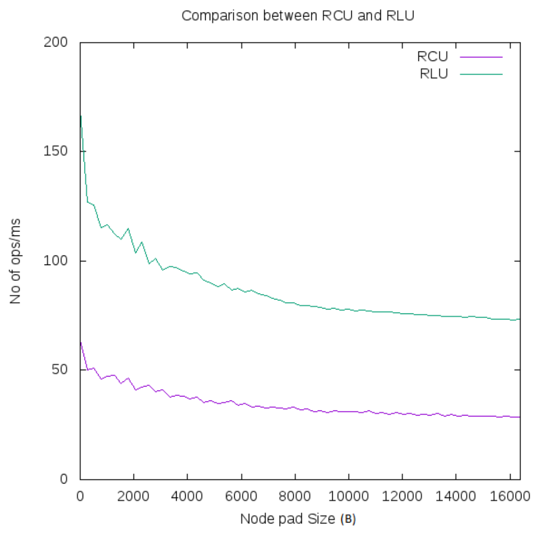

Figure 8.7: Throughput for RCU and RLU Linked lists with 20% updates

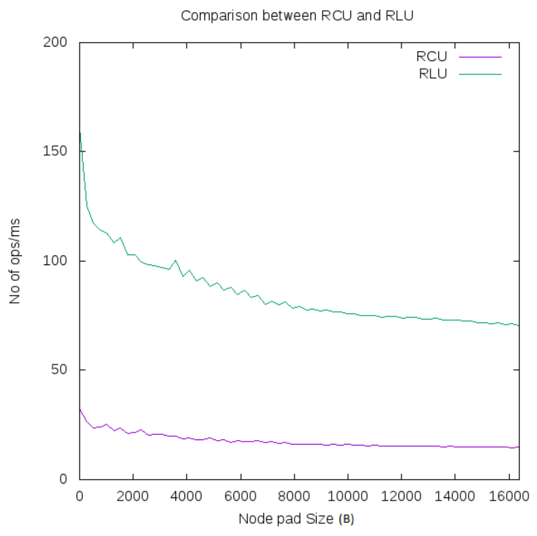

Figure 8.8: Throughput for RCU and RLU Linked lists with 40% updates

As can be seen form the above plots, the throughput decreases with increasing node size as expected but node size has no effect on the relative performance of RCU and RLU mechanisms. Thus node size may not be considered as a parameter for comparing the above lock free mechanisms when using with linked lists.

While linked list may be good for various scenarios, hash list finds its use when you need to quickly find or insert a data structure based on hash-key. Hash List node size thus do play a important factor for optimizing the hash list performance. Though the above optimization requires some other considerations, this part tries to understand the relative performance of RCU and RLU protected hash lists with varying node size and fixed number of threads.

The following plots shows the experimental results for determining the relative performance of the two lock free mechanisms with increasing node size and with varying update rate for a simple hash list with 10000 initial insertions and fixed number of threads.

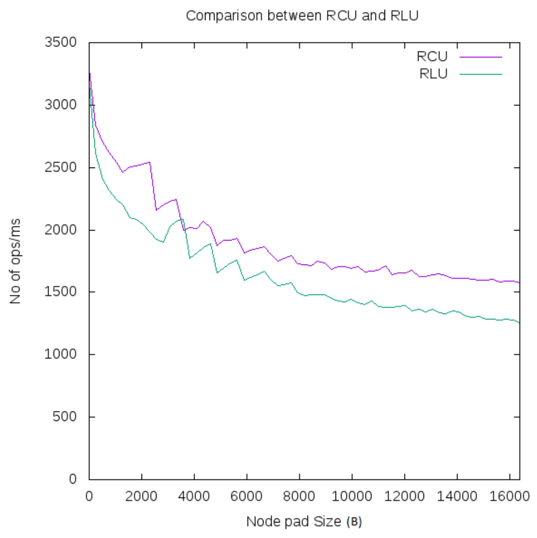

Figure 8.9: Throughput for RCU and RLU Hash lists with 20% updates

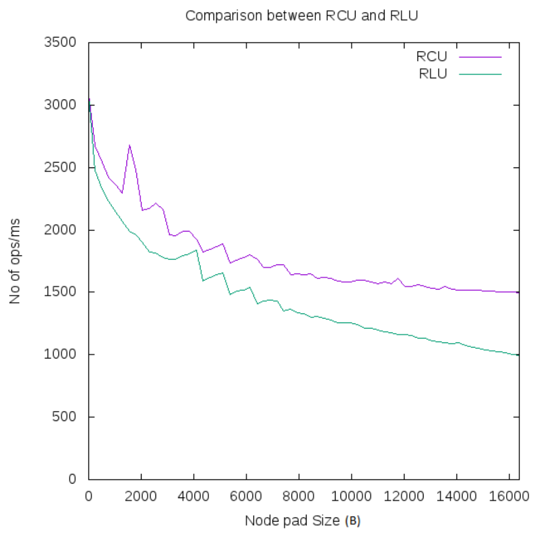

Figure 8.10: Throughput for RCU and RLU Hash lists with 40% updates

As seen in the case of linked lists, node size doesn't effect the relative performance of hash lists either. Thus node size may not be considered as a parameter for comparing the above lock free mechanisms when using with hash lists.

8.2 Kernel Side of the Story

While the above section deals with a synthetic benchmark for comparing two lock free mechanisms that work on read mostly data structures, this section deals with the writer behavior in a RCU protected data structure in the Linux Kernel under a given workload. The writer behavior that this section tries to explore are the following:

- The first goal is to determine the read-write ratio and the write lock contention for the busiest RCU protected data structure in the Linux Kernel. This behavior will lead to understand if the writer requires any optimization to lower the contention for different update rates.
- The second goal is to determine the usage semantics of the list based design of RCU protected data structures, i.e to understand if the lists are used as a set or as a queue. This will enable us to address if certain changes are required semantics such as building a per core list and help make a design that best fits the usage semantics.

The next part of this section addresses the experiments conducted for measuring the writer contention and also the write to read ratio. This will help us in understanding if any optimization is required for the helping the writers.

8.2.1 Measuring the Writer Lock Contention per core

To measure the lock contention of the writer, the problem is divided in to first finding the busiest data structure protected by RCU and then profiling the same data structure with the same workload. The entire design can be shown in Figure 8.11.

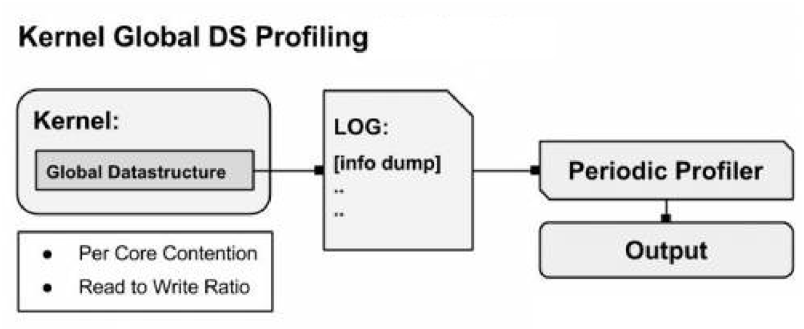

Figure 8.11: Design for Profiling a RCU protected Data Structure

As can be seen, the profiler is used to parse the log and calculate the most accessed data structure or per core contention periodically. The entire work is done in two steps:

- Firstly, the kernel source is modified to log every RCU list update call. The workload is run and the log is later parsed to get hold of data structure that is accessed the most.
- Secondly, the kernel source is again modified to add time checks across the writer locks and log the difference with the current core id. The time checks are done using RDTSCP calls to ensure that the time checks are found without much overhead. The workload is run and the log is periodically read and parsed to find the level of contention in that period of time per core. The parsed result is later used to get a periodic plot.

There are two experiments are done using a synthetic workload that creates random threads and a web-server workload with multiple clients. This two workloads keep track of a RCU protected per process linked list and a global linked list The following are the two workloads and the data structures they aim to track:

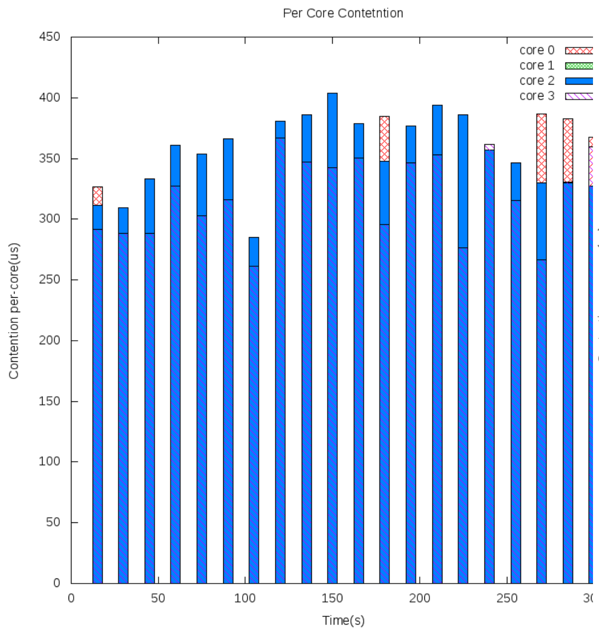

Figure 8.12: Per Core Contetnion

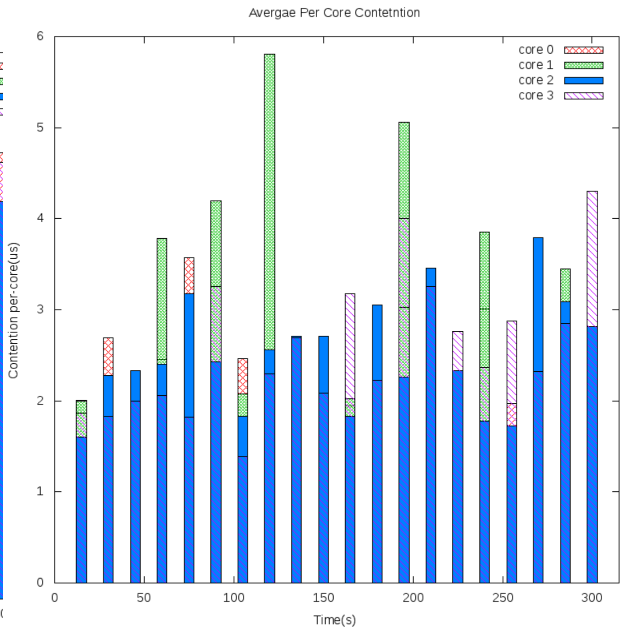

Figure 8.13: Average Per Core Contention

- **Tracking the list of all processes:** This data structure contains the list of all processes(task_struct list) and is updated during creation and termination of the process. The synthetic stress workload to keep track of it creates 250 threads each creating 500 empty threads in parallel every 5 seconds. The workload was run on a system with 4 cores Intel i5 processor(1.2GHz) and 4 GB memory.

- **Tracking the epoll linked list:** This data structure consists of the list of file descriptors for serving multiple clients. The workload consists of a nginx local server hosting a simple website and has 10000 parallel clients requesting 1000 parallel requests every 10 seconds. The local server runs on a system with 4 cores Intel i5 processor(1.2GHz) and 4 GB memory.

The two workloads are run independently to keep track of the RCU protected list data structures respectively. The experiments done tries to stress the data structure usage to find out the usage behavior of the lock free mechanism in use.

The first experiment is run for tracking the list of all processes as mentioned above. The log is parsed every 15 seconds to generate the periodic plots shown below. The plot, Figure 8.12 shows the time(converted from the number of cycles) wasted every period of 15 second of running the workload with more than 95% updates made using the stress workload.

The second experiment uses the web server workload mentioned above that uses epoll to server multiple requests and the log is parsed every 15 seconds for 5 minutes duration to generate the periodic plots below.

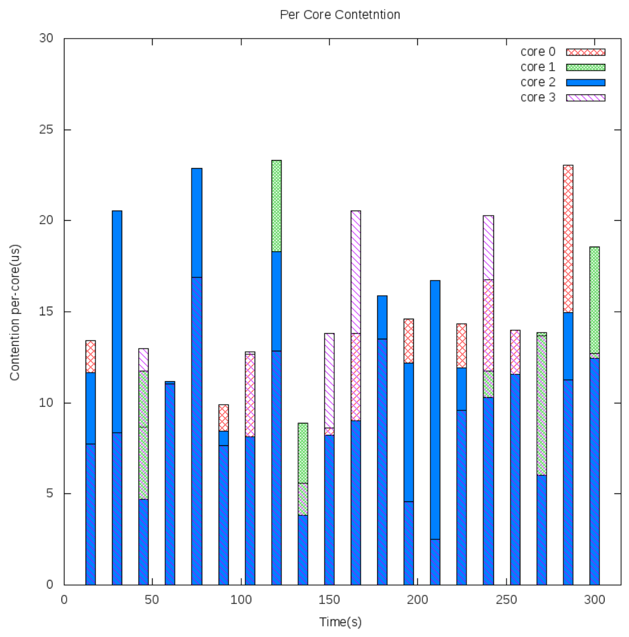

Figure 8.14: Per Core Contetnion

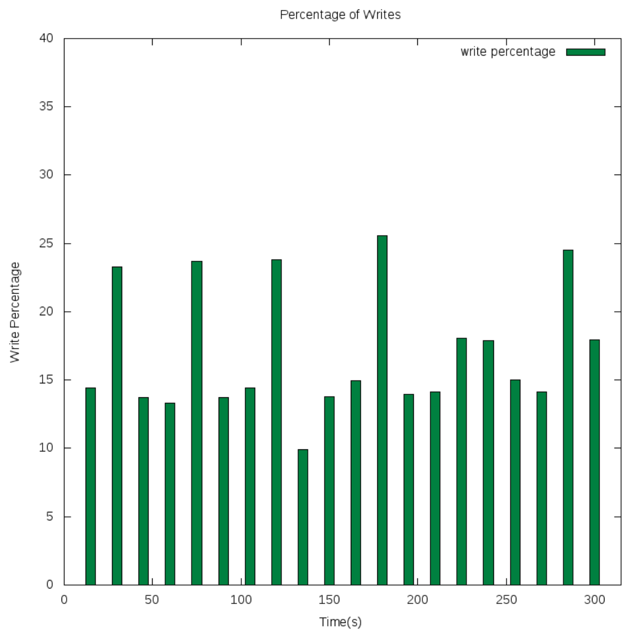

Figure 8.15: Write Percentage

As can be seen in the above experiments, the contention with the workloads measured in terms of cycles or time wasted is quite low. Even with stressing the system with multiple updates in parallel, the overall contention and the average contention in the previous experiment(Figure 8.13 is desirable. Thus it can be said from the above experiment that a web server as seen will scale well with the use of RCU protected epoll list.

Some more experiments will be done in stage 2 that includes a large scale workload and then continue the above experiment to find the link between contention and update rate. The low contention shows why RCU have had a significant effect on the Linux Kernel. The other thing to lookout for is the relation between write-read ratio and the contention.

8.2.2 Usage semantics of RCU protected linked lists

This part focuses on understanding the usage semantics of a RCU protected data structure. We are currently focusing on linked list and will deal with hash-lists in stage 2. To understand the usage semantics of the updates in the RCU protected linked lists, the rcu api calls are monitored to understand the usage scenarios.

The following usage behaviors are monitored:

- Ordered insertion of the RCU protected list.
- Ordered deletion from the protected list.

From a log of rcu api calls in the Linux Kernel[5], we find the usage statistics of data structures currently protected by RCU. The following table, Table 8.1, lists the statistical information regarding the list based data structures in use.

Usage Statistics	**Linked List**	**Hash List**
No of such RCU protected data structures in the Kernel	236	71
Inserts at list tail	109	–
Inserts at head	127	71

Table 8.1: Usage Statistics of RCU protected Data Structures

From the above table, it can be seen that most of the data structures inserts only at one end of the list. The next step is to identify the usage behavior of such data structures such as deletion behavior will help us determine if we can make some optimization to the list based design. In our case study we have taken a few data structures and tried to answer the above set of questions that is summarized in the below table, Table 8.2.

Name of List Head	Insertion Behavior	Deletion Behavior
List of process with same thread-id	Inserts at head	Delete anywhere
List for epoll file descriptors	Inserts at tail	Delete anywhere
List of all running processes	Inserts at head	Delete anywhere

Table 8.2: Usage Behavior of RCU protected Linked Lists

The lists maintain no ordering and the additions are mainly done at one end, while the deletions happen anywhere. This property can be used to build a design that divides the list based data structure and pins each of them to a particular core so that each core has access to its own per-core list. This may optimize the cache performance and some more experiments and work must be done in stage 2 on this part.

8.3 Design of the RCU protected Data Structure

This section addresses the design of the current RCU protected data structures and questions the use of certain semantics. In the current Linux Kernel, most of the used data structures uses list based semantics. We will mainly focus on RCU protected data structures such as linked lists and hash lists as other tree data structures are not yer supported by the RCU mechanism.

Linked lists are used widely in the Linux kernel and as can be seen from the above section, [pro num] linked lists are protected by RCU. Hash-lists on the other hand uses linked lists simply for chaining each object that have the same hash-key. Though the list based semantics are widely popular due to its O(1) insertion and deletion time, it doesn't provide a cache friendly way for traversal. This section tries to show that if arrays were used in place of lists small relatively small node sizes, the throughput would have been better due to improved cache performance.

The experiment conducted below uses the benchmark used for comparing RCU and RLU which randomly inserts/delete linked list with a given update rate. In this experiment the linked list RCU implementation was compared with an RCU protected array implementation with fixed node size of 16 bytes. The below plots shows the results with different update rates.

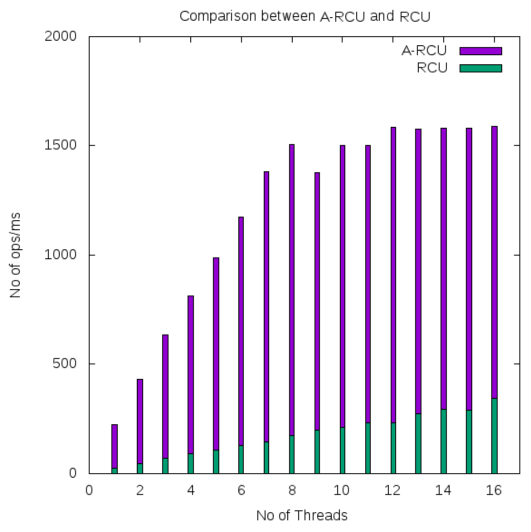

Figure 8.16: Comparing ARCU and RCU Linked lists with no updates

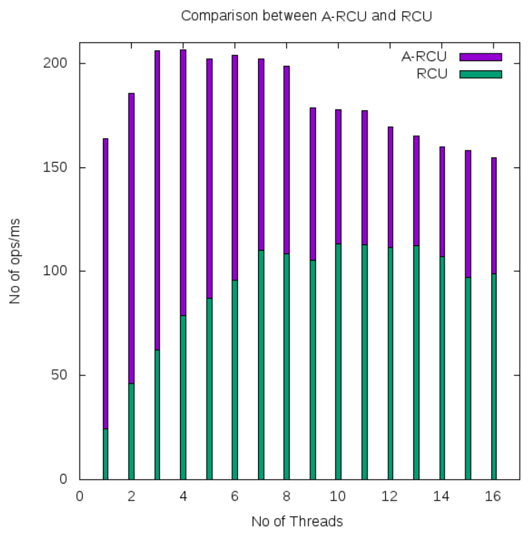

Figure 8.17: Comparing ARCU and RCU Linked lists with 20% updates

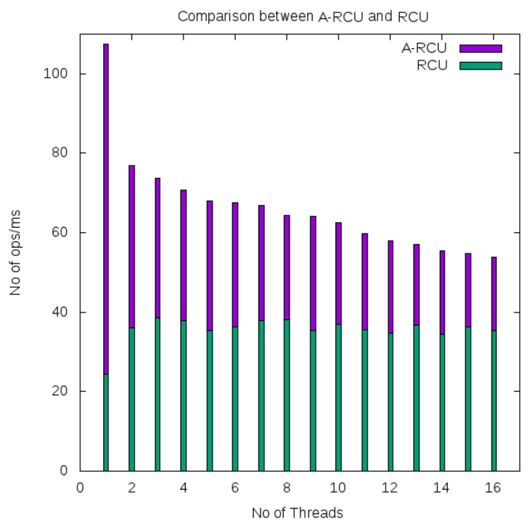

Figure 8.18: Comparing ARCU and RCU Linked lists with 60% updates

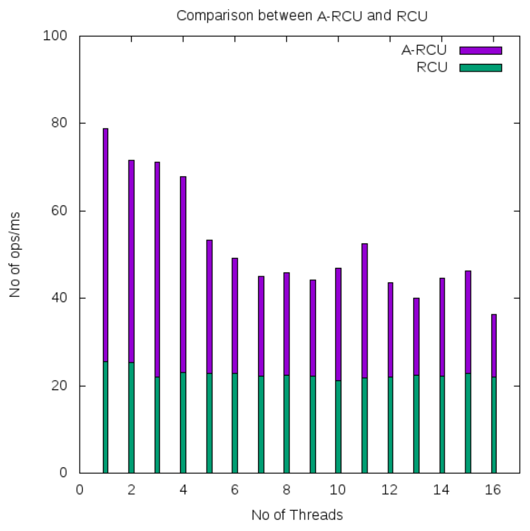

Figure 8.19: Comparing ARCU and RCU Linked lists with 100% updates

Though the plots shows earlier shows RCU protected arrays to have better throughput than linked lists, the node size must be considered. The following small experiment with varying node size and fixed update rate of 20% writes shows that the array based semantics is good upto a certain node size as seen in plot(Figure 8.20).

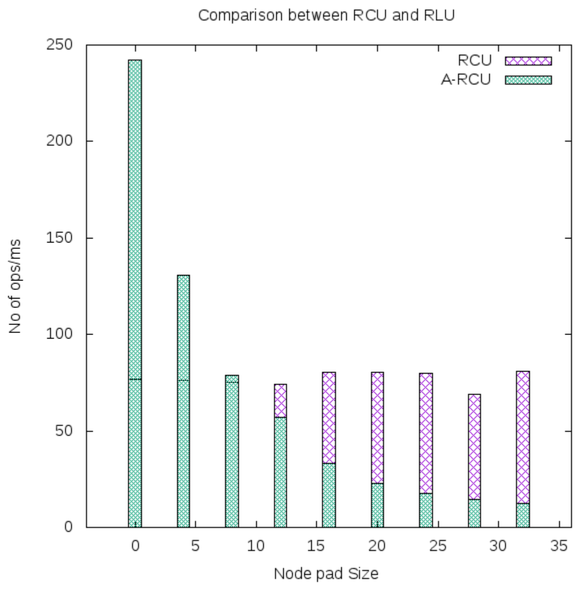

Figure 8.20: Effect of Node Size on Performance of ARCU and RCU

This section will be explored a bit further in stage-2 probably with an array based data structure design that is cache friendly and also doesn't loose track with large node size.

Chapter 9

Conclusion

In this report, a few questions have been answered while others remain open for more exploration. In this stage, we have seen how the mTCP stack can be used to port multi-threaded master-worker applications and proved its correctness. The major question now arises is about the scalability of such master-slave design with user space stacks which leaves this project open for future research. As per our performance evaluation, the design of master worker model using non blocking I/O is not at par with the single threaded model.

We also explored how the RCU mechanism effects the writers in the Linux Kernel and what is the read to write ratio with a real life workload such as a simple web-server. We have explored only the contention when RCU is used with Linked Lists and have seen how using a simple RCU protected array may prove better than RCU protected lists with small node size. Another thing we have seen was that, node sizes do not play a very big role when comparing the already existing lock free mechanisms.

Acknowledgments

I would like to thank my adviser, Prof. Sriram Srinivasan for helping me figuring out how to proceed with the given problem and to work out a good design. I would also like to thank my co-advisor, Prof. Purushottam Kulkarni for the long discussions that we had during the design phase of the project.

Bibliography

[1] Cliff click's non blocking queue: http://www.azulsystems.com/blog/cliff/2007-04-23-nonblocking-hashtable-source-code. 2007.

[2] Sunghwan Ihm Dongsu Han EunYoung Jeong Shinae Woo Muhammad Jamshed Haewon Jeong and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore system. In *11th USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[3] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, October 2001.

[4] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. 2015.

[5] Paul E. McKenney. Rcu linux usage log: http://www.rdrop.com/ paulmck/rcu/linuxusage/linux-4.3.rcua. 2015.

[6] Paul E. McKenney. Some more details on read-log-update: https://lwn.net/articles/667720. 2015.

[7] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, October 1998.

[8] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15, June 2004.

[9] Dominik Scholz. A look at intels dataplane development kit. In *Network Architectures and Services*, August 2014.

[10] John D. Valois. Lock-free linked lists using compare-and-swap. , *Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, pages 214–222, August 1995.