NETWORK STACK OPTIMIZATION IN MAPREDUCE

Samantha Miller

Department of Computer Science
University of Texas at Austin

Committee: Simon Peter (Advising), Vijay Chidambaram, and Etienne Vouga

May 2018
Abstract

Recently, improvements in networking hardware have caused traditional assumptions about network bottlenecks to become invalid. In environments with very fast networks such as datacenters, the networking bottleneck has shifted from networking hardware to network processing done in the CPU. One proposed solution to this is kernel bypass networking in which network stacks execute in user-space, avoiding overheads and inefficiencies in the Linux kernel. Tools such as DPDK and mTCP implement kernel bypass and provide application-ready interfaces. This project explores using mTCP and DPDK in MapReduce to analyze the practicality and effect of using kernel bypass in datacenter applications. Performance results weren’t exemplary, mostly due to difficulty integrating the tools. It’s likely that more impressive results could be achieved in the near future with some more work in the area and more mature tools.

Contents

1 Introduction 3
2 Background 3
  2.1 Networking 3
  2.2 MapReduce 4
3 Overview of DPDK and mTCP 4
  3.1 DPDK 4
  3.2 mTCP 5
4 Overview of MapReduce Lite 5
  4.1 Network Structure 5
    4.1.1 Sockets 5
    4.1.2 Signaling Queue 6
    4.1.3 Message Format 7
    4.1.4 Copies 7
5 Design 7
  5.1 Incorporating Kernel Bypass 7
  5.2 Reducing Copies 8
6 Evaluation 9
  6.1 Total Runtime 10
  6.2 Application Runtime 10
  6.3 Sleep Overhead 12
  6.4 Network Stack Performance 13
  6.5 Removing Copies 15
7 Experiences 17
8 Related Work 17
9 Future Work 18
10 Conclusion 18
1 Introduction

Recent trends in computing hardware have forced operating systems developers to rethink many design choices and question long-held assumptions. In particular, general knowledge has been: CPUs are fast; almost everything else is slow, particularly networking and storage. In this model, it’s acceptable for processing of information from devices to put load on the CPU because the I/O device is the bottleneck. With modern networking hardware, this assumption is invalidated, and now, CPU intensive packet processing means that applications aren’t able to reach the full potential of their networking hardware.

Recent work [21] has shown that most processing time is spent in the Linux kernel and that significant performance gains can be achieved using user-space network stacks instead, among other potential optimizations. My project explores these changes and their possible benefits for MapReduce, a distributed programming paradigm that provides the concept behind modern data processing applications. This acts as a proof-of-concept for datacenter user-space networking and can provide useful details for future work in the area.

This project specifically uses mTCP [12] and DPDK [11]. DPDK implements per-packet processing in user-space, and mTCP builds on top to provide a fully user-level network stack with a similar interface to Linux sockets. Ideally, this would mean that very few modifications would need to be made to an application to use kernel bypass: attaching network cards to DPDK and switching Linux socket calls to mTCP socket calls. This was mostly true, but these steps weren’t completely trivial and several workarounds needed to be added to avoid bugs within mTCP. These workarounds significantly hurt performance, causing the modified version of MapReduce in this project to perform worse than the baseline. If these issues were fixed, it’s likely that kernel bypass networking would give performance benefits, especially for Mappers and when many small messages are sent.

The next section will discuss background of networking and MapReduce. Sections 3 and 4 provide relevant details about the specific tools used for this project. Section 5 explains the optimizations that were made, and section 6 analyzes the performance results. Experiences are described in section 7. Related work is described in section 8, and future work to the project in section 9. Section 10 concludes the report.

2 Background

2.1 Networking

Hardware development has been accelerating quickly recently, especially for networking. Recent cards provide both significantly faster throughput and new potential uses than those of the past. Today, many datacenters have wired connections of between 10 and 100 Gbps throughput. On top of that, new functionality has been added to networking hardware, including programmable network cards and switches [20]. This allows the device to contain a small amount of code and state so basic packet processing can be done in the hardware.

We can use information about modern networking to calculate a budget for packet processing: how long packet processing can take for network hardware to still be the bottleneck. Given 10 Gbps network cards and a minimum packet size of 64 bytes, each packet must be processed in 6 nanoseconds in order to achieve line rate. If 100 Gbps network cards are used, each packet only gets 0.6 nanoseconds. A kernel crossing takes 200 nanoseconds [21], so doing even one takes too much time. For larger packets, this is less of a concern. If packets are 256 kilobytes instead, the kernel crossing takes less than one percent of the processing budget, so can be done without issue.

The introduction of new networking hardware and the pressure put on these network stacks has inspired new designs for removing operating system overheads from the network path. The main idea behind this is kernel bypass, or bypassing the kernel during network processing. Moving the network stack to user-space provides several benefits. Primarily, the expensive cross into the kernel is avoided. Also, some functionality can be removed from the network stack. A user-space network stack doesn’t need the same security checks as the kernel because it can’t execute privileged instructions. In addition, the kernel network stack must be very generic, providing the many layers of functionality that an application may desire. A user-level network stack can be designed to more closely match with the application, providing only what the target application needs. Because of these reasons and other changes made to the network stack, implementations of kernel bypass networking generally remove much of the per-packet overhead.
Two implementations that are receiving wide attention are RDMA [22] and DPDK [11]. RDMA implements kernel bypass by requiring hardware support in network cards, such as Infiniband [10]. DPDK is a specific optimized per-packet processing stack. It’s implemented in software as a set of user-space network card drivers and incorporates many different optimizations. This project focuses on DPDK because it doesn’t require changes to networking infrastructure, while RDMA does.

2.2 MapReduce

MapReduce is a programming model created at Google and published in 2004 [5]. It defines a general and high level approach to programming distributed data analysis. In this model, data analysis is broken down into two main phases borrowed from functional programming: the Map phase and the Reduce phase, with the Shuffle phase in between. Each phase is implemented by worker nodes called Mappers and Reducers, respectively. In the Map phase, the Mapper receives input key/value pairs, processes these, and generates intermediary key/value pairs. The Shuffle phase afterward moves these key/value pairs from the Mappers to the Reducers, though this work is often done by the Map or Reducer workers. In the Reduce phase, Reducers combine all intermediary key/value pairs with the same key into one key/value pair and output this to a file. Each Reducer task is in charge of a section of the key-space and processes all intermediary key/value pairs in that section. If the Reduce function is commutative and associative, a Combiner can also be provided that does part of the Reduce work at the Mapper and outputs the resulting key/value pairs. The specific Map, Reduce, and Combine functions are written by the user; MapReduce itself mainly provides the infrastructure for the computation. The traditional example application is word count. In order to implement this, the Map phase takes in lines of files and outputs (word, 1) tuples for every word in every line. The Reducer then combines all of these together to generate (word, number of instances), and the output files will contain the number of times each word shows up in the input files.

The scalability and parallelism provided by MapReduce have made the idea popular since its creation. Still, in the 14 years since MapReduce was first introduced, the specific tools used in data processing have changed. Now, applications that grew out of MapReduce are used more commonly than MapReduce itself. These programs, such as Apache Hadoop [2], Apache Spark [28], or Google’s Cloud Dataflow [1] and Flume-Java [4], use many of the same ideas as MapReduce and so are affected by many of the same problems and can benefit from the same solutions. This project focuses on MapReduce because its simplicity makes it easy to modify and its wide influence clearly displays the potential wide-reaching effects of the project.

Because distributed algorithms must send information between machines, their performance is limited by the speed of the networking stack in the systems they’re deployed in. MapReduce is especially affected by networking speed because all intermediary key/value pairs must be moved between the Mappers and Reducers. This problem is only getting worse as data-intensive applications, like machine learning, become more prevalent. Advances in I/O hardware will help mitigate MapReduce’s performance issues, but that depends on software being able to take full advantage of the increased performance capabilities and new functionality. For this project, I analyze networking in MapReduce, studying the effects on performance of recent networking developments, among other possible network stack optimizations.

3 Overview of DPDK and mTCP

3.1 DPDK

DPDK (Data Plane Development Kit) [11] is a library providing fast packet processing. It’s implemented as a set of highly optimized user-space drivers and a generic interface to access these drivers. Its impressive packet processing speeds are due incorporating many different techniques for performance enhancement, from polling to reducing copies and beyond. These performance gains are especially noticeable for small messages, which are handled inefficiently by the Linux kernel. DPDK does make some tradeoffs, such as needing a core to consistently poll on the network card, so it’s primarily useful in environments where packet processing is a bottleneck.

This high performance is achieved through a myriad of specific optimizations. Placing the drivers primarily in user-space means that they rarely have to make the expensive context switch into the kernel. DPDK also avoids the overhead of interrupts by constantly polling on the network card. While this busy waiting
takes more CPU than blocking for an interrupt, it is much faster. Using statically allocated memory pools for buffers greatly decreases the amount of time spent on memory allocation, and backing the memory with huge pages reduces TLB cache misses. These buffers are mapped into shared memory space so DPDK can be zero-copy. Lockless queues increase efficiency by reducing wait time. Packet header prefetching allows for fast packet classification. Altogether, these many techniques for optimizations along the packet processing path combine to create a highly performant library.

3.2 MTCP

DPDK itself only provides a packet processing interface; the TCP and IP stacks in the kernel aren’t modified. This comes with the same downsides of using the kernel packet processing, namely the complexity of a generic library and the overhead of system calls. MTCP [12] acts as a solution to this issues. It’s a scalable user level TCP stack especially designed for multicore systems. It builds upon lower-level optimized networking structures, such as DPDK or Packet Shader’s IO engine PSIO [9], only adding a complementary user-space implementation of the TCP and IP protocols. In benchmarks, mTCP has proved itself to be significantly faster than the traditional Linux network stack.

The library uses several techniques to optimize the TCP stack. The user-space implementation avoids the overheads of switching to the kernel. Instead of acting purely as a library and only executing behavior when the application requests it, mTCP starts a separate thread at creation to ensure that TCP’s timing assumptions aren’t broken. To improve multicore scalability, mTCP uses per-core data structures and requires that the application choose a core to process networking for the duration. These data structures are lock-free, small, and pull from a statically allocated memory pool so access and allocation can be fast and cache misses minimized. Like many optimized applications, mTCP also batches handling of events to reduce the context switching overhead. Creating separate queues for control packets and using statically allocated buffers increase mTCP’s performance for short-lived connections, a situation it’s optimized for. Along with these changes, mTCP provides a simple interface that is almost identical to Linux sockets and epoll for easy adoption.

4 Overview of MapReduce Lite

The project used and modified MapReduce Lite [17], an open source implementation of MapReduce. This specific version was used because of its simplicity and ease of deployment. MapReduce Lite is written in C++ and can run on top of personal machines without any modification, using the local file system. Mappers and Reducers are processes running a Python wrapper around the core C++ implementation. Scheduling is simple because task locations are specified by the user and the processes are started as soon as the application is run, with the Reducer started before the Mappers to avoid race conditions.

The library provides two modes of execution, batch and incremental. Batch processing is the typical structure of MapReduce where intermediary key/value pairs are written to files on disk and these files are copied to the reducer. Using incremental processing, each tuple is sent from the Mapper to the respective Reducer as soon as it is generated. In batch processing, a Python wrapper copies the files from the Mappers to the Reducers after the Map phase is done, but the incremental processing mode uses Linux sockets within the C++ networking structure. Because mTCP is provided as a C library and because incremental mode better reflects modern datacenter networking use, this project uses incremental mode. The rest of this section will focus on the network architecture used in MapReduce Lite.

4.1 Network Structure

4.1.1 Sockets

This application has a fairly simple network protocol and uses a small number of sockets. Each Mapper has a unidirectional send connection to each Reducer, and each Reducer has the corresponding receiving connection, plus the listening socket. Because the Reducers have no information to send to the Mappers and don’t acknowledge messages from the Mappers, bidirectional connections aren’t created. If there are $M$
Mappers and $R$ Reducers, the total number of connections is $M \times R$ and the total number of created sockets is $2 \times M \times R + R$.

### 4.1.2 Signaling Queue

Figures 1 and 2 display the network structure of MapReduce Lite. To send messages, MapReduce Lite uses a queue, called a signaling queue. When the Map worker is ready to send a key/value pair to the Reducer, it “sends” by adding this message to a destination specific queue, blocking if the queue is full. In another thread, the application is waiting for the sending socket to be ready (using libevent [15]). When the event triggers, that thread attempts to remove a message from the queue, blocking if there is no message, and then sends that message along the associated socket. The Reducer works analogously, except that data received from the socket can contain multiple messages. In this case, the received buffer is broken into individual messages before being added to the queue. The Reduce worker then reads from that queue and processes the contained messages.
The signaling queue is implemented as a statically allocated circular buffer. Signals are used to share information about the state of the queue between producers and consumers. Internal conditions are used to block and unblock the queue when it’s full or empty. If a thread needs to block on a condition, it waits on it. If another thread removes from or adds to the queue, it signals the respective condition to indicate that state has changed. The queue itself can also be signaled by a producer when that producer is finished adding to the queue. Consumers can use this to check if all data have been handled. The queue also locks around adding and removing, providing thread safety. This likely has little effect on performance because there are generally few producers and consumers.

4.1.3 Message Format

Each message includes one key/value pair. The message first contains the key size then the value size, both measured in bytes and of type `uint32`, followed by the key and value. The key and value can be any arbitrary string of bytes but must fit within a message buffer. The size of this buffer is configurable but defaults to 32MB + 2 * `sizeof(uint32)`. In practice, messages are usually much smaller than this. For the large word count application tested in this project, the average message size was around 13.5 bytes.

4.1.4 Copies

Copies are one of the main targets of network stack optimization. Often, messages are copied around several times before being sent on the network, incurring overhead from both significant memory accesses and context switching to the kernel to allocate space for these copies. MapReduce Lite is no exception to this, and copies data many times in its network stack. In the life cycle of a key/value pair, from origin at the Mapper to processing at the Reducer, the data is copied seven times.

On the Mapper's side, the key and value are first copied (1) to create the actual message in the correct format. The application then adds the message to the signaling queue, using a copy (2), and the sending thread copies it out again (3). This version is then sent on the socket, incurring any copies within the Linux TCP stack. The Reducer then receives up to 4K bytes from its socket into a buffer. Because this buffer may contain multiple messages concatenated together, each message is copied into its own buffer (4) and added to the signaling queue (5). The application thread then removes the message from the queue (6) and copies the key and value into local variables (7).

Most of these copies are because of the design of the signaling queue. This structure gives the queue control over its own memory, particularly meaning that it can statically allocate a buffer at initialization and know that no other code will touch this memory. This is a very safe design, but isn’t optimal for such a network dependent application.

5 Design

In this project, several techniques were considered to improve the network performance in MapReduce Lite. MapReduce itself sends large amounts of data between the Mapper and Reducer, so it’s a good candidate for studying network improvements. Specifically, the incremental mode of MapReduce Lite sends every key/value pair as an individual message, so the speed of network processing is likely a significant portion of the total application run time.

The main concepts were employing kernel bypass networking and reducing the number of times data is copied within the application. Other approaches such as offloading the network stack to programmable network cards, using hardware queues to replace software ones, and batching sends were discussed. It was decided that these wouldn’t be as likely to produce significant results within the timeframe of the project, but they could be implemented as future work.

5.1 Incorporating Kernel Bypass

Much of the overhead of network processing comes from the Linux kernel \[21\]. Kernel bypass networking avoids this by moving the network stack into user-space, both reducing the number of context switches needed for system calls and allowing the network stack to be more tailored to the application. This design
decision, along with other optimizations made within the libraries that implement it, greatly increases the performance of the network stack. Because MapReduce Lite sends many small messages and is network limited, it should especially see gains from avoiding the Linux kernel network stack. This project uses mTCP and DPDK, networking libraries that together move the network stack fully into user-space. Because mTCP implements a full TCP stack, this project doesn’t benefit from the customizability of kernel bypass networking, but was simpler to implement that writing a MapReduce Lite specific network stack.

This project modifies MapReduce Lite to use mTCP sockets instead of Linux sockets with DPDK providing the lower levels of the network stack. Using DPDK involves removing the desired network card from its kernel driver and attaching it to the associated Poll Mode Driver provided by DPDK. Depending on the network card involved, this ranges from straightforward to impossible. As one of its optimizations, DPDK expects the system to have a stash of huge pages. When it’s started, DPDK allocates all of the provided huge pages for its buffers, meaning that only one MapReduce worker process can be started per machine at a time.

mTCP is designed to be easy to use, so its API is very similar to that of Linux sockets. This meant that a fairly small amount of code needed to be changed to use mTCP sockets, mostly to switch the API calls and to keep track of the mTCP context. As an additional change, the networking event mechanism also needed to be modified. MapReduce Lite originally used libevent to wait for its sockets to be ready. Libevent acts as a generic event mechanism, choosing between platform specific system calls, such as Linux’s epoll, to implement waiting. Because the sockets were changed, the socket event mechanism needed to be updated to match. mTCP itself provides an alternative in the form of its own epoll functions. Changing libevent to use these function proved prohibitively difficult, so the calls to libevent were replaced with mTCP epoll calls within MapReduce Lite. While this simplification could have given a slight undeserved performance boost to the kernel bypass version, this wasn’t noticed in the experiments.

mTCP itself isn’t a very mature tool, so several bugs were encountered throughout this project. The listening socket needed to be set as non-blocking before accepting connections or it would never unblock after the connection was successfully created. This change was mirrored in the Linux socket based version and isn’t likely to affect performance. A more significant work-around was the need for adding sleeps to avoid race conditions. If the mTCP provided epoll functionality was used too soon, events would never be received. To fix this, three second sleeps were added to every process before starting the polling mechanism. Because this is only done once per process, it shouldn’t affect a production scale job significantly, though it was noticeable in this project. Another issue occurred per send and receive that caused the network rate to need limiting. If messages were sent or received too quickly, the connection would be dropped without warning, likely due to a race condition within mTCP. Sleeps of 100 microsecond per send and 2 milliseconds per receive were added. Because this slowed the connections down to much less than the maximum throughput of the setup, this severely reduced the performance of the system.

5.2 Reducing Copies

Another common way to increase performance of network limited applications is to reduce the number of times messages are copied. Memory copies are expensive and network stacks often do them many times if the designers aren’t paying specific attention. DPDK and mTCP focus on limiting the amount of data copied, so employing kernel bypass already makes this optimization within the TCP stack. This still doesn’t address the copies done within the application as it sends and receives packets.

MapReduce Lite does a large number of these, adding seven copies of each piece of data per message. Four of these copies per message are from interactions with the signaling queue on both sides of the connection. This queue only benefits the structure by decoupling network operations and application processing, allowing the application to continue if the network isn’t operating at the same speed as data processing. However, there is very little time when the network isn’t ready and the whole MapReduce run still has to block until all data is able to be sent along the connections. With faster kernel-bypass networking, there should be little need to execute network commands in the background. In addition, waiting for an event before using the sockets doesn’t provide extra safety guarantees; sends or receives from the socket will block if the connection isn’t ready. This queue, then, seems to cause more issues than it solves and could be entirely removed to improve performance of the application.

While this approach wasn’t implemented due to a lack of time, the possibility was thoroughly explored.
and possible performance gains were evaluated. Changing MapReduce Lite to directly send and receive from sockets instead of the queue would be fairly simple. The application doesn’t batch sends, so the Mapper just replaces adding to the queue with the provided socket sending code with no other changes. The Reducer is slightly more complicated because multiple messages can be received at once. These are split and added to the queue separately so the application can easily receive one message at a time. In order to remove the queue, the code that receives from the queue would need to be modified to handle blocks of messages. This change would likely be fairly straightforward. The effect of these copies on runtime was measured, so while this change wasn’t implemented, the performance gain by removing the queue can be estimated fairly accurately.

6 Evaluation

These experiments used three nearly identical machines, each with an Intel Xeon CPU E3-1220 v5 processor with 4 cores, 4 or 8 GB of RAM, and at least two network cards: one Broadcom Corporation NetXtreme BCM5720 card used for management and process start-up and an Intel Corporation 82599ES 10-Gigabit card used in the tests. All network cards are connected to a 100 Gbps switch using 10 Gbps Ethernet cables. For the baseline tests, the Intel 10 Gbps cards were connected to Linux interfaces and given IP addresses used in the tests. For the modified MapReduce Lite tests, the cards were attached to the DPDK ixgbe driver and given the same IP addresses through network interfaces.

![Diagram of test configuration]

Figure 3: The test configuration

As shown in Figure 3, each of the three machines ran one MapReduce Lite process for a total of two Mappers and one Reducer. Tests were started by running a startup script on one of the Mapper machines. This script copied the necessary binaries and input files to the target machines and created the necessary processes.

Three different test structures were run. In the first, very small files were given to the Mappers, and a total of 36 messages (18 from each Mapper) were sent to count the 36 words. The other two tests counted the words in *Hamlet* and *Romeo and Juliet*, for a total of 52384 words. The first of these tests used a Combiner with the Mapper, so only 12983 messages were sent to the Reducer (7656 and 6313 from each Mapper, respectively). The last test didn’t use a Combiner so all 52384 messages were sent (31984 and 25879, respectively).
6.1 Total Runtime

The above graph illustrates the total runtime of each test, using both the unmodified versions of MapReduce Lite and the version using kernel bypass. As can be seen from the graph above, this specific integration of kernel bypass significantly increases the total runtime of the MapReduce Lite jobs in these tests, and the slowdown grows as the number of messages sent increases. Given that the kernel bypass test execution was artificially slowed to avoid race conditions, it’s not surprising that these tests both take more total time and slow down as messages are sent. The following sections examine this performance degradation and attempt to explain its origin.

Additionally, the runtime of the baseline tests doesn’t increase with the number of messages. This is likely because the tests run were much smaller than the expected size in a datacenter so networking wasn’t the overhead. Indeed when analyzing the test runs themselves, we see that each process takes less than one second to execute, so most of the displayed runtime comes from setting up and tearing down the tests. Larger tests were considered but decided against because they wouldn’t be likely to provide new information to the comparison given the overheads seen in these fairly small tests.

6.2 Application Runtime

<table>
<thead>
<tr>
<th>Messages Sent</th>
<th>Total Time (s)</th>
<th>Queue Time (s)</th>
<th>Calculated Application Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>14.42</td>
<td>3.00</td>
<td><strong>11.42</strong></td>
</tr>
<tr>
<td>12983</td>
<td>15.64</td>
<td>4.10</td>
<td><strong>11.54</strong></td>
</tr>
<tr>
<td>52384</td>
<td>19.23</td>
<td>7.73</td>
<td><strong>11.50</strong></td>
</tr>
</tbody>
</table>

Table 1: Effect of network slowdown on application runtime in kernel bypass

The above table represents, for the Reducer, the total time spent in the signaling queue compared to the total runtime. The Reducer is used because it is started before the Mappers and finishes after, so its total runtime best reflects the total time spent running the MapReduce Lite job. Because network operations are executed in a separate thread than the application, it’s difficult to record the exact effect of network modifications on the application runtime. The time spent in the signaling queue is a good approximation
because it represents the time that the application spends receiving data, including any waiting for data to be received from the network. The baseline version of MapReduce Lite spends much less than a second in the signaling queue for any test size, so it can be inferred that the vast majority of time spent in the queue is due to network stack modifications, most likely the sleeps but potentially other sources as well.

From Table 1, it can be seen that this increased network stack time totally accounts for the message-dependent performance degradation. Therefore the change to kernel bypass sockets doesn’t add any other significant sources of message-dependent overhead. Later sections will further discuss this increase in time spent in the queue to determine the source.

![Reducer Task Time](image)

**Figure 5: Time Reducer spent in application processing**

However, some overhead is still unaccounted for. As seen in Figure 5, the time spent in the application doesn’t grow a large amount, but the total runtime (excluding time spent in the signaling queue) still increases by about 5 seconds. Therefore, there must be another source of overhead during application setup or teardown. It’s most likely that this comes from the time spent creating the sockets and connections while the application is being started. Both mTCP and DPDK need to be started when the socket is created, and this setup involves a decent amount of logging to the screen, so it’s not surprising that this would add overhead. Because this is a one-time overhead, it would likely be dominated by application runtime in a production environment.
6.3 Sleep Overhead

The added sleeps greatly decrease the network speed, forcing the 10 Gbps network to operate at 2 Mbps at the Reducer and 130 Kbps at the Mappers. Because receives from the signaling queue block until something is in the queue, the time the Reducer spends in the queue is greatly affected by this manual slowing of the network rate. Figure 6 relates the time spent in the queue and the time of the Reducer's rate limiting sleeps. Given that these two values are so close and generally grow with each other, it’s reasonable to assume that sleeps entirely account for the extra time spent in the queue. Similar analysis can be done for the Mappers, though they’re less affected by the sleeps because data processing is also done between each send. From this, we see that if the race condition in mTCP were fixed so that the sleeps were no longer needed, there would likely be no performance degradation beyond setup time. In this situation, the speed of the network stack itself would be able to impact the performance.

![Figure 6: Time spent in signaling queue vs time spent sleeping for kernel bypass Reducer](image-url)
6.4 Network Stack Performance

![Hamlet Mapper Network Time](image)

**Figure 7:** Time first Mapper (*Hamlet*) spent sending

![Reducer Network Time](image)

**Figure 8:** Time Reducer spent receiving
To analyze the maximum potential performance gained by switching to a kernel bypass networking stack, it makes sense to consider only the effect of sends and receives along the connections, ignoring the sleeps. As done so far, we first consider the Reducer because its performance closely correlates to the performance of the whole job. As seen in Figure 8, the Reducer’s networking performance is improved slightly, though it decreases as the number of messages grows. The networking time for the smallest test is reduced by 84%, down to an insignificant change in the largest test. As the tests send more messages, the performance gain decreases, likely because more data has been sent along the connection so the receive processes more data. The Linux kernel is least efficient when processing small messages, so it’s reasonable that it would catch up to kernel bypass in performance as the message size grows.

Therefore, to see the full benefit of kernel bypass, we have to analyze the Mapper. Since both Mappers are similar, only the first, the one that processed Hamlet, is shown here. Because all of these messages are small, the modifications made in this project greatly decrease the amount of time spent sending packets. From the smallest test to the largest test, the modified performance stack reduces network send times by 74%, 59%, and 57%. This would correlate to some extra performance increase in the Reducer as it would be able to receive more data at a time, but this effect is difficult to quantify.

![Theoretical Hamlet Mapper Time](image)

Figure 9: Theoretical runtime for an optimized kernel bypass Mapper
Using these numbers, it’s possible to calculate the theoretical performance gain from a kernel bypass worker without artificial slowdown. The runtimes for the unmodified MapReduce Lite are calculated by adding together the application processing time and the time spent in the signaling queue, thus the total amount of time spent on that worker’s task from start to finish. Because the networking time would be reflected in the queue time, the theoretical performance of a kernel bypass worker can be calculated by subtracting the unmodified worker’s network time from the runtime and adding the respective kernel bypass worker’s network time. Values calculated in this manner are shown in Figures 9 and 10. The networking rates calculated from this method are still less than 10 Gbps, so it’s reasonable that these results could be achieved given the experimental setup. From these graphs, we see that the Reducer’s performance is mostly unchanged, but the Mapper gains a modest boost, reducing total time by up to 17% for the tests ran in this project.

6.5 Removing Copies

While the signaling queue wasn’t removed from MapReduce Lite and all of the memory copies stayed in the code, we can analyze the possible performance gain of removing the queue using the same method as above. This can be done by simply subtracting the total time spent on signaling queue copies from the total application time. Because some time is spent processing in the queue, this is likely somewhat of an underestimate of the performance change. A different approach would be to simple replace the signaling queue time with the network time, but this would overestimate any performance gains by not taking into account time spent waiting on the network. Because this waiting time is likely more significant than the queue processing time, the first approach is used.
Figures 11 and 12 show this calculation. Here, both the Mapper and Reducer see slight performance increases: up to 7.5% for the Mapper and 10% for the Reducer. Both worker types are affected fairly evenly because both do the same amount of in-queue copying per message. While this change doesn’t appear dramatic for these tests, it would be quite significant as the size of the data being processed grows and the
total amount of messages being copied increases with it. Because of time constraints and the relatively small
gains, the signaling queue, and so the copying, wasn’t removed from the implementation.

7 Experiences

The design and progress of this project have been necessarily influenced by the specific tools used. While
these had many useful properties that made this project possible, there were also significant issues along the
way that hindered development. MapReduce Lite seems to have been abandoned and hasn’t been noticeably
updated within the past four years. This meant that running the application was difficult and several bugs
had to be fixed along the way.

This project also required attaching network cards to their DPDK driver and having them communicate
with each other. Because DPDK is implemented as a set of network card drivers, any progress made on this
was specific to each type of network card used in the project. Our research cluster includes four different
kinds of network cards: Intel 82599ES 10 Gbps, Intel XL710 40 Gbps, Mellanox ConnectX-5, and Netronome
Agilio CX. Last year, previous students discovered that the Netronome cards weren’t compatible with our
machines and so couldn’t be used in tests. As part of this project, we attempted to connect the Mellanox
cards to DPDK. These cards are supported, so this should have been possible, but was abandoned after
several weeks without progress. Both of the types of Intel cards were easily compatible with DPDK, so these
became the focus of the project. While the two types of cards could operate successfully during the test
application, when MapReduce Lite was used, messages sent by one type of card were never received by the
other type. The final solution was to order enough Intel 10 Gbps cards to run the experiment totally on
these cards. Additionally, the version of DPDK included in mTCP used functionality that was deprecated
in the Linux kernel version 4.8. The kernel needed to be downgraded to version 3.16 for the libraries to run.
Since then, mTCP has switched to the next version of DPDK.

While mTCP was designed to be easily integrated with existing applications, this design choice didn’t
seem to extend into the implementation itself. It wasn’t always clear how to modify the function calls to use
mTCP and internal error handling was all but nonexistent. Several modifications were made to the project,
such as making the listening sockets non-blocking and changing the event delivery mechanism, in order for
mTCP to work. MTCP was also changed slightly, fixing a bug that caused connections to be rejected if the
SYN packet was received too soon. As evidenced by the added sleeps, the final product used in the project
wasn’t yet bug free.

8 Related Work

The network overhead of MapReduce is discussed and optimized in several other papers. Mellanox UDA [19]
uses RDMA [22] to improve the performance of Hadoop MapReduce and achieves significant performance
results. Another work [26] also considers designing Hadoop to use RDMA over Infiniband [10] network cards.
This project does similar work using DPDK and mTCP and so doesn’t need to modify the network API and
infrastructure as much as RDMA would.

Several other works explore application-design methods of reducing network usage in MapReduce. [8] and
[27] optimize MapReduce networking by changing the scheduling algorithm. [25] designs a network-optimized
merge phase, reducing the time before the Reduce task can start. These works and other attempts to improve
MapReduce performance by modifying the behavior of the application are mostly orthogonal to this project.

The original idea of removing the network stack from the kernel seems to have originated from the
Exokernel [6], an operating system that implemented almost all abstractions in a user-space library operating
system. [7] discusses writing a user-space network stack on top of an exokernel OS. This implements similar
behavior to mTCP and DPDK and so doesn’t need to modify the network API and infrastructure as much as RDMA would.

LibZero [16] and netmap [23] are libraries similar to DPDK in that they provide fast packet processing.
LibZero focuses on zero-copy packet processing while netmap provides several of the same optimizations as
DPDK though is still partially in the kernel. DPDK is more focused on multicore scalability and has wider compatibility with other libraries.

[18] removes barriers between the application and the network so network can be optimized for the specific use cases of that application. DPDK presents an API that allows a network stack to be build on top of it, so users can do similar optimizations. This project didn’t design a MapReduce specific network stack due to lack of time and instead used mTCP to provide an implementation of TCP.

Arrakis [21] uses SR-IOV [14] network virtualization and a library OS to optimize data plane network operations. Like this project, Arrakis responds to changes in network hardware by moving the network stack to user-space, though the two works use different network tools. The IX [3] operating system addresses a similar problem as Arrakis but doesn’t fully remove the kernel from data plane management and uses different network hardware techniques.

An extension to the idea of hardware-enabled network processing changes, FlexNIC [13] proposes installing packet processing rules into a programmable network card to further remove the CPU from the network path. Ideas in this vein using SmartNICs [20] were considered but weren’t possible within the timeframe of the project.

9 Future Work

As mentioned earlier, several optimizations were considered that weren’t implemented due to a lack of time. Reducing the number of copies is discussed most in this report because it would be fairly easy to implement and would likely give noticeable performance benefits. Other possibilities include using programmable network cards, writing a MapReduce specific network protocol, and batching sends.

Programmable network cards could provide especially interesting results. This would involve putting at least part of the MapReduce network stack logic onto the network card so packets could be processed in hardware. This would involve a significant amount of work, designing the network protocol so it can be handled by hardware, but could give significant performance benefits. That project would act as a proof-of-concept of using programmable NICs to accelerate network heavy applications.

10 Conclusion

This project analyzed the effects of implementing MapReduce using an optimized kernel-bypass network stack. In order to understand the potential benefits and pitfalls of doing this in a commercial database system, a production-ready kernel bypass stack was used. Despite this, the implementation still took some effort and expertise to finish.

Difficulties with integrating the network stack caused performance results to be unimpressive, but this will likely change as the tools mature and internal bugs are fixed. The results do show that kernel bypass stacks have potential to accelerate common network-dependent distributed applications. With more implementation work in these areas, it would be likely that datacenter systems could significantly increase network throughput and decrease the latency for distributed applications using the techniques presented and suggested in this project.
References


