Replication of Concurrent Applications in a Shared Memory Multikernel

Yuzhong Wen

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Application

Binoy Ravindran, Chair
Ali R. Butt Co-Chair
Dongyoon Lee

June 17, 2016
Blacksburg, Virginia

Keywords: State Machine Replication, Runtime Systems, Deterministic System, System Software

Replication of Concurrent Applications in a Shared Memory Multikernel

Yuzhong Wen

(ABSTRACT)

State Machine Replication (SMR) has become the de-facto methodology of building a replication based fault-tolerance system. Current SMR systems usually have multiple machines involved, each of the machines in the SMR system acts as the replica of others. However having multiple machines leads to more cost to the infrastructure, in both hardware cost and power consumption. For tolerating non-critical CPU and memory failure that will not crash the entire machine, there is no need to have extra machines to do the job. As a result, intra-machine replication is a good fit for this scenario. However, current intra-machine replication approaches do not provide strong isolation among the replicas, which allows the faults to be propagated from one replica to another.

In order to provide an intra-machine replication technique with strong isolation, in this thesis we present a SMR system on a multi-kernel OS. We implemented a replication system that is capable of replicating concurrent applications on different kernel instances of a multi-kernel OS. Modern concurrent application can be deployed on our system with minimal code modification. Additionally, our system provides two different replication modes that allows the user to switch freely according to the application type.

With the evaluation of multiple real world applications, we show that those applications can be easily deployed on our system with 0 to 60 lines of code changes to the source code. From the performance perspective, our system only introduces 0.23% to 63.39% overhead compared to non-replicated execution.

This work is supported by AFOSR under the grant FA9550-14-1-0163. Any opinions, findings, and conclusions expressed in this thesis are those of the author and do not necessarily reflect the views of AFOSR.

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays semiconductor industry has pushed the CPU core count to a historically high level. Having a computer system with high CPU core count and large memory capacity is cheaper than ever before. However with the technology advances, we still cannot ignore the fact that computer systems suffer from transient failures time to time. Given a fix probability for the failure of one CPU core, adding more CPU cores leads to a higher probability of full system failure. Current SMP operating systems the failure of a single core can bring down the entire system. To be able to recovery from such severe failures, having backup machines is always a good idea. When the primary machine fails, the backup machines are set to be able to take over the previous work and carry on the services.

With the idea of replication, there are a good amount of works that allow users to have multiple machines to act as replicas [1] [2] [3] [4] [5] [6]. However, there are kinds of faults that would fail part of the OS or just the application, without breaking the entire machine. In this case full machine replication is not needed, as a result, people have tried to discover replication solutions inside the single machine, which we classify them as intra-machine replication. In [7] and [8], the authors proposed replication systems that can have a redundant execution instance along with the original one. But doing the replication inside the same OS still cannot mitigate a transient fault that could fail the OS itself. To have a full stack replication that can minimize the impact of an OS failure, several works have investigated the approaches of doing replication via virtualization [9] [10] [11], in order to provide stronger isolation of the replicas. However, such solutions can still suffer from the faults that could crash the hypervisor. Moreover, the hypervisor based techniques usually require the replication of the entire VM state, which brings more overhead.

To achieve resilient fault-tolerance replication inside a single machine, we have to provide even stronger isolation for all the replicas. Multi-kernel [12] [13] provides the idea of running multiple OS kernels on a multi-core system without the support of a hypervisor. On a multi-kernel OS, each kernel has its own dedicated CPU cores and a part of the physical memory, the kernels can communicate with each other via messages. While multi-kernel

OSes are designed to explore new means of extracting multi-core performance for concurrent applications, we found this kind of OSes can provide strong isolation that fits perfectly for intra-machine fault-tolerance replication. The Popcorn Linux [13] project is a research effort aimed at building scalable systems software for future generations of heterogeneous and homogeneous multi/many-core architectures. As a multi-kernel OS, Popcorn Linux is able to host multiple Linux instances simultaneously on a multi-core machine. Building a replication system that utilizes multiple Linux instances on Popcorn Linux is the perfect solution for doing intra-machine replication. In this intra-machine replication model, we can treat each kernel instances as a machine node and try to adopt inter-machine replication techniques onto Popcorn Linux. Moreover, with this level of isolation, the transient fault on one kernel instance can hardly get propagated to others.

State machine replication (SMR) has been widely used for inter-machine fault-tolerance replication. In SMR, it models the services to be replicated with a set of inputs, a set of outputs and a set of states. The replication system ensures that for a given input set, from the same initial state, the replicas can produces the same state transition which in turn leads to the same output. Such a system is able to be resilient to failures in one or more replicas (depends on how many replicas are there in the system). To provide such property, determinism is required for the state machine, otherwise the state of state machines will diverge in different states even with the same input set. However, most of current SMR systems need to model the state machine into the applications explicitly. For existing applications, especially concurrent applications, it is a changellenge to transparently model the states of application for replication.

With the explorations above, we see the following challenges for doing intra-machine replication:

- Is it possible to adopt inter-machine replication techniques to intra-machine replication? Will we have the same performance charactristics?
- How to transparently model existing concurrent applications for SMR replication?
- While having less fault coverage than inter-machine replication, can intra-machine replication have less overhead?

In this thesis, on top of Popcorn Linux, we have built an intra-machine SMR system to replicate concurrent applications. In our system, we have a kernel instance as the primary replica and another kernel instance as the secondary replica. As an SMR system, we model the state of a concurrent application by its system call output and thread/process inter-leavings. We implemented two different replication modes, that are originated from previous inter-machine replication systems, to synchronize the thread/process inter-leavings across replicas. By doing so, we make sure that the replicated concurrent applications can have consistent outputs on both primary and secondary. Moreover, our system is transparent to applications, existing software can be deployed on our system with minimal modification.

1.1 Contributions

Coressponds to the challenges we raised previously, this thesis presents the following contributions:

- To synchronize thread inter-leavings of replicated concurrent applications, based on Popcorn Linux, we implemented two different replication modes in the kernel. Both of them are originated from previous inter-machine replication solutions. Two replication modes have achieved the same goal in two different directions: Deterministic Execution uses a deterministic algorithm to decide the order of execution on both primary and secondary; while Schedule Replication enforces the secondary replica to follow the non-deterministic execution order that happened previously on the primary kernel.

- Our system provides a common programming interface for both replication modes. By wrapping a code section with our __det_start and __det_end system calls, the execution order of wrapped sections can be the synchronized on both primary and secondary kernel. In order to support existing applications transparently, based on this common interface, we also implemented a set of runtime supports that allows the user to run applications on our system with minimal code modification.

- To explore the pros and cons for both replication modes, and the cost of our intra-machine replication, we evaluated different types of concurrent applications on our system. For computational application we had maximum 63.39% slowdown for Deterministic Execution and maximum 36.3% slowdown for Schedule Replication. For two web servers we had maximum 25.22% slowdown for Deterministic Execution and maximum 1.96% slowdown for Schedule Replication. With the evaulation we observed that Schedule Replication is the better replication mode for intra-machine replication.

1.2 Scope

This thesis mainly discusses about how to ensure the same thread/process inter-leavings across the replicas, and some key system calls to ensure the consistent state of server applications. We do not address the non-determinism from signal, file system, random number generator and memory address space. For the replication of server applications, a member of SSRG has already implemented the replication of the TCP stack and connection recovery from kernel failures, which was able to support replicating simple single-threaded server applications prior to this work. On top of that, the work described in this thesis provides the ability of replicating real world multi-threaded/multi-process server applications.

1.3 Thesis Organization

This thesis is organized as follows:

Chapter 2 presents the background of Popcorn Linux, which is the multi-kernel system we are using for building our intra-machine replication.

Chapter 3 presents our first replication mode, Deterministic Execution.

Chapter 4 presents our second replication mode, Schedule Replication.

Chapter 5 presents the additional runtime support that we implemented to eliminate some residual non-deterministic issues, and a runtime library to simplify the application deployment on our system.

Chapter 6 shows the performance evaluation of our system on multiple concurrent real world applications.

Chapter 7 concludes the work and discusses some future works.

Chapter 2

Popcorn Linux Background

Our replication prototype is built on top of Popcorn Linux [13]. It is a multi-kernel OS which allows a multi-core system to boot multiple Linux kernels. In this project, we leverage this architecture to achieve inter-machine replication. The replicated applications will run on different kernel instances and Popcorn Linux provides strong isolation of the resources on the machine.

2.1 Hardware Partitioning

In Popcorn Linux, hardware resources are partitioned into arbitrary divisions, as shown in Figure 2.1, each booted kernel instance can have the full control of its own partition. In general, CPU cores, memory and devices can be splitted into different partitions, each kernel can have its own dedicated CPU cores , memory and devices. The hardware partitioning provides a very strong isolation for all the kernels and the applications running on them, which is ideal for our intra-machine fault tolerance model. Especially when the partition is done based on NUMA zones, a critical hardware error happens on one kernel's hardware partition can hardly get propagated to another.

2.2 Inter-Kernel Messaging Layer

Popcorn Linux comes with a high efficient messaging layer for inter-kernel communication [14]. From the implementation perspective, the messaging passing is basically just a set of memory copy operations between the address space of the sender kernel and receiver kernel. As a result, the cost of sending a message is very low (under 5us). The replication system developed in this project heavily leverage Popcorns messaging layer.

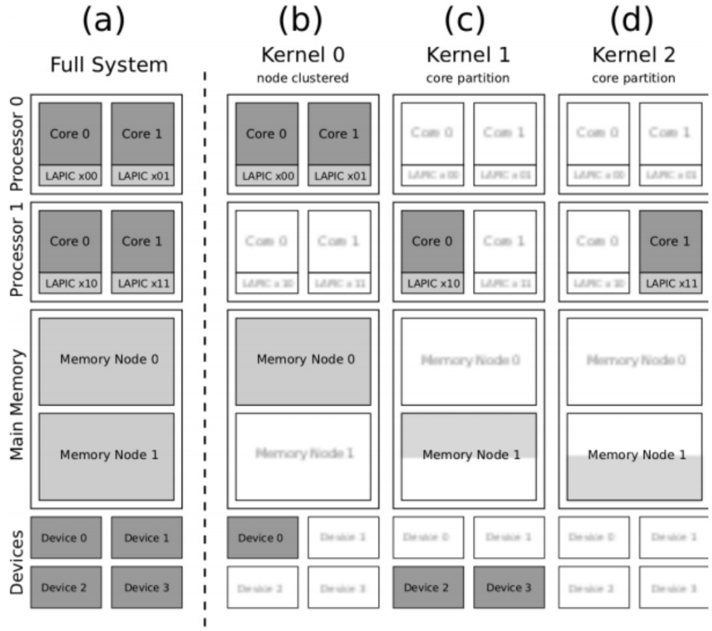

Figure 2.1: Popcorn Linux hardware partitioning

2.3 Popcorn Namespace

In Linux, namespaces are used for creating isolated execution environment for applications. Popcorn Linux implemented a new Linux namespace in order to isolate normal applications and replicated applications. Applications inside Popcorn Namespace will utilize our replication service, once the user enters the Popcorn namespace, all the applications that run inside it will be replicated to the secondary kernel.

2.3.1 Replicated Execution

When a user launches an application inside the Popcorn namespace, Popcorn Linux will gather all the information for launching this application (full executable path, environment variables, etc) and send it to the secondary replica. On the secondary, a kernel thread will be spawned with all the information it received, to create an identical process as the one in the primary replica. This makes launching the replicated application on the secondary kernel to be transparent to the user.

2.3.2 FT PID

In order to match the replicated tasks, Popcorn Linux introduced an extra field to Linux's task_struct called ft_pid. As shown in Figure 2.2, ft_pid is an array of IDs in order to match the tasks on both sides, other parts in the system use ft_pid to synchronize the execution of tasks on both sides with the same ft_pid. Popcorn Linux makes sure that the creation of a process/thread is deterministic on both sides, so that all child processes/threads can be matched properly on both sides.

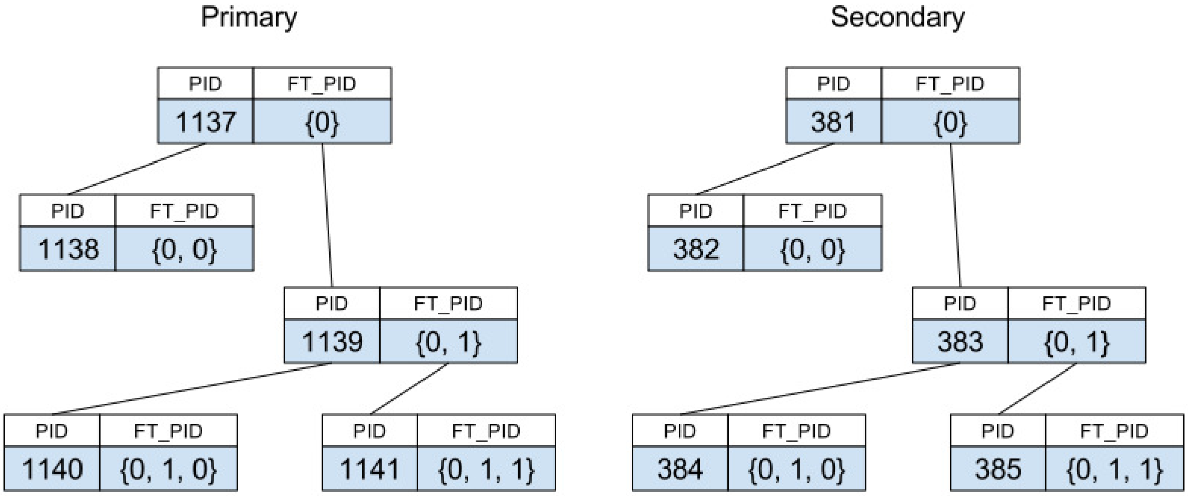

Figure 2.2: An example of ft_pid

2.4 Network Stack Replication

The TCP stack in Linux is intrinsically non-deterministic. Given the same network stream, the TCP stack might end up with different states with different runs. For example, for read/write operations, the TCP stack may return a non-predictable number of bytes to the application reading from the socket, which might in turn leads to different states during the replication.

Based on the idea of FT-TCP [4], Popcorn Linux comes with a network stack replication technique to eliminate this kind of non-determinism for replicating network applications. On the inbound path of Linux's network stack, all the TCP traffic that are related to an active Popcorn namespace will be copied to the secondary and be put into a TCP logging buffer. All read/write/accept/listen/close system calls that are related to a replicated socket will be synchronized between two kernel instances. Any event that will cause the TCP state to change will be replicated inside the stacks(SYN, FIN, etc) of all the kernels. While for the events that do not affect the TCP state (normal read and write), the secondary simply bypasses its own TCP stack and uses the TCP logging buffer to create the same output for those system calls.

In general, the network stack replication service provides following important functionalities for network applications replication and fault recovery:

- The internal state of every replicated socket is synchronized.
- Read and write operations can have consistent output across all replicas, in terms of size and content.
- Upon accept is returned, it is made sure that both replicas are returning the same socket with the same TCP stream.
- Upon the primary is crashed, with the consistent state of TCP stack, the secondary can take over the network device driver and continue with previously connected connections.

Chapter 3

Shogoki: Deterministic Execution

A deterministic system provides a property that given the same input, a multithreaded program can always generate the same output. Such a system fits perfectly for our replication purpose. As long as the primary and secondary receive the same input, the replicated application will sure end up with the same state and generate the same output.

For multi-threaded programs, an observation is that as long as the threads don't communicate with each other, the execution is sure to be deterministic[15]. For example, in pthread based programs, all the inter-thread communications are synchronized by pthread primitives. By making the interleaving of sychronization primitives to be deterministic, the entire program is sure to be deterministic. With this observation, some runtime deterministic solutions actually enforce determinism by trapping pthread primitives[16][17][18]. This type of deterministic system is called ”Weak Deterministic System”. It assumes that the applications are data race free, and only guarantee the deterministic interleaving of thread synchronization primitives such as mutex locks and condition variables. Our implementation falls into this category.

In this chapter we present Shogoki[1]: Deterministic Execution. This replication mode maintains a global execution order, according to this order, an execution token is passed among all the tasks deterministically. Only the task with the execution token can enter the deterministic area, and the token will be held on this task only if it leaves its deterministic area.

3.1 Logical Time Based Deterministic Scheduling

Inspired by Kendo [18] and Conversion [19], this scheduling policy maintains a logical time for each task inside the current Popcorn namespace. There is a ”token” being passed among

[1] Means Unit 1, in Japanese

all the tasks in the namespace according to the logical time of each task. Our system provides following system calls for the applications to control the thread-interleaving:

- __det_start: When it is called, only the task holds the token can proceed. If the current thread is able to proceed, this thread will be marked as ”in a deterministic section”.
- __det_end: When it is called, the system will increase the current thread's logical time by 1, and marks it as ”out of a deterministic section”.

.

The token is updated whenever the logical time is changed, and it is passed based on following rules:

- Among all the tasks inside the namespace, the one with the minimal logical time gets the token.
- If multiple tasks have the same minimal logical time, the one with the smallest PID gets the token.

.

Figure 3.1 shows an example use of the system calls. Simply wrap pthread_mutex_lock with __det_start and __det_end will make the acquisition of the mutex to be deterministic.

If the logical time is updated but the one has the minimal logical time is sleeping in __det_start, the one whose updates the tick will wake the sleeping one up. As long as the replicated application updates logical time in a same way on both primary and secondary, they will sure end up with the same thread interleaving. Figure 3.2 shows a simplified version of this algorithm (some mutual exclusion points are omitted here).

To make an application to run in a deterministic way, one should put __det_start and __det_end around the synchronization primitives such as pthread_mutex_lock , so that the order of getting into critical sections is controlled under our deterministic scheduling.

3.1.1 Eliminate Deadlocks

With wrapping all the pthread_mutex_lock with our deterministic system calls, there is a potential risk of having deadlocks. Serializing all the lock acquisitions with our implementation basically means putting a giant global mutex lock around every lock acquisition. As shown in Figure 3.3, Thread 2 has a lower logical time and try to acquire the mutex(b), however mutex(b) is contended, as a result Thread 2 will call futex_wait and put the thread into sleep until mutex(b) is released by someone else. At this point, Thread 2 will never increase its

```

void producer() {
    while (running){
        item = generate_item();
        syscall(__NR_det_start);
        pthread_mutex_lock(mutex);
        syscall(__NR_det_end);
        putItem(queue, item);
        pthread_mutex_unlock(mutex);
    }
}

void consumer() {
    while (running){
        syscall(__NR_det_start);
        pthread_mutex_lock(mutex);
        syscall(__NR_det_end);
        item = getItem(queue);
        pthread_mutex_unlock(mutex);
        consume_item(item);
    }
}
```

Figure 3.1: An example use of the deterministic syscalls

```
void __det_start()
{
    if (token->token != current)
        sleep(current);
    current->ft_det_state = FT_DET_ACTIVE;
}
void __det_end()
{
    current->ft_det_state = FT_DET_INACTIVE;
    __update_tick(1);
}
void __det_tick(int tick)
{
    __update_tick(tick);
}
void __update_tick(int tick)
{
    current->tick += tick;
    token->task = find_task_with_min_tick(ns);
    if (is_waiting_for_toUponken(token->task))
        wake_up(token->task);
}
```

Figure 3.2: Simplified implementation of deterministic system calls

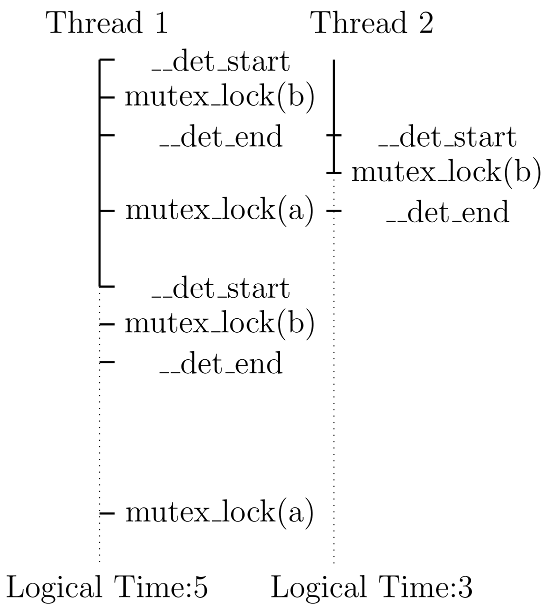

Figure 3.3: An example of deadlock

logical time until mutex(b) is released. So Thread 1 will never goes through the __det_start, and it will never unlock mutex(b) which means Thread 2 will never be woken up.

Since we already know that a contented mutex will call futex_wait [20] to wait for a unlock event, the solution to this deadlock problem is to temporary remove the thread in futex_wait out of the deterministic schedule, and add it back when it returns from futex_wait. In the example of Figure 3.3 Thread 1 will be able to proceed its __det_start and keep executing. In order to not to break the determinism, we guarantee the following:

- We guarantee that the waiting queue in futex_wait is strictly FIFO, which means the wakeup sequence will be the same as the sequence of getting into futex_wait. Since the latter one is ensured by our __det_start, with this hack to futex, the wake up sequence from futex_wait will be the same sequence determined by previous __det_start. This is implemented by fixing the priority of each futex object, so that the priority queue inside futex_wait can behave like a FIFO queue.

- We guarantee that when waking up from a futex_wait, the thread always waits for the token before returning to the user space. With this implemented, the timing (in terms of logical time) of getting out of a contended pthread_mutex_lock will be deterministic. This is implemented by adding a __det_start after the wake up point of futex_wait.

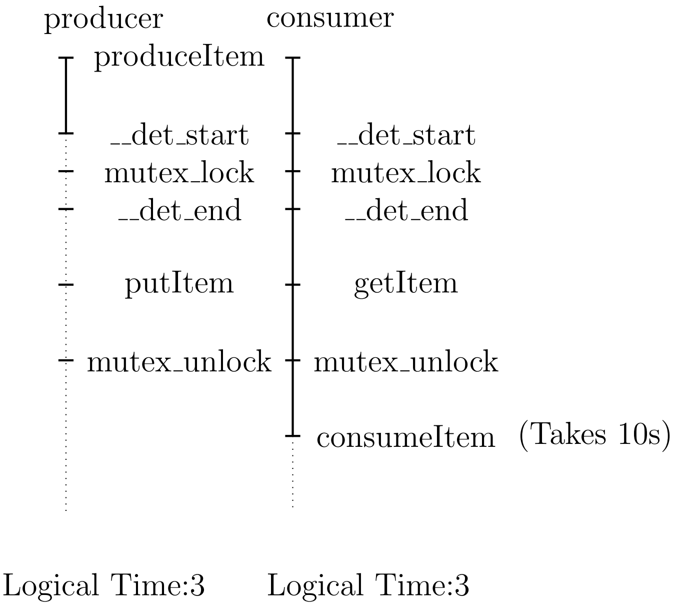

Figure 3.4: An example of logical time imbalance.

3.2 Balance the Logical Time

Only increasing the logical time by 1 at __det_end isn't enough. With an example we show how this could break the scalability and how to mitigate this problem. In Figure 3.4, we show a particular execution point of the producer-consumer model in the program snippet we presented in Figure 3.1, solid lines represents the path that is already executed. In this case, consumer reaches consumeItem with logical time 3 and has the token. Assume the real execution time of consumeItem is 10s, which means that when the consumer reaches __det_end, it would be at least 10s later, that is, the producer has to wait at __det_start for at least 10s. However we've already enforces the access order of the mutex, the execution out of the critical section should go in parallel since threads don't communicate at that point, in worst case, this kind of waiting will turn a parallel program into a serial program. Figure 3.5 shows an extreme example where pbzip2 becomes a serial program with unbalanced logical time, it doesn't scale at all as we increase the thread count.

Generally, logical time imbalance can happen in two cases:

- A task is running for a long time (in user space).
- A task is sleeping for a long time (in kernel space).

In the upcoming sections we will discuss the solution of each of the cases.

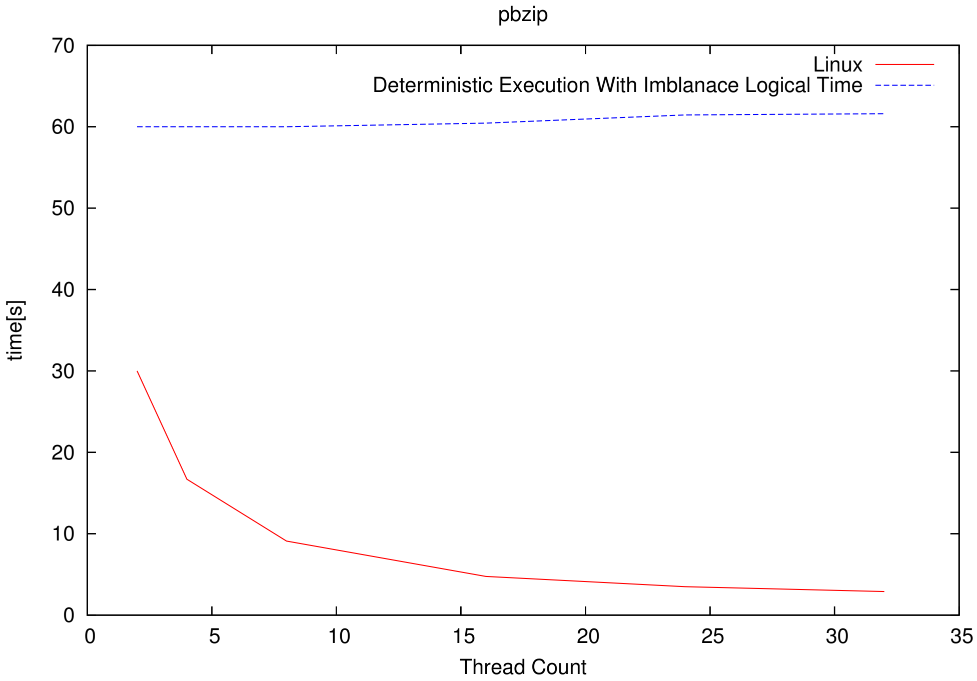

Figure 3.5: pbzip2 without logical time balancing

3.2.1 Execution Time Profiling

When a task is running in a computational region (in user space) which might take a long time, the logical time of the task should increase along with the execution. In Kendo this is done by counting retired read instructions using performance counters to track to progress of a running task and increases its logical time accordingly. However it is hard to ensure that on the primary and the secondary the performance counter can have the same behaviour [21], as a result we have to find another way to track the progress of a running task.

Instead of deciding the logical time during the runtime, we discovered a way to settle the logical time during the compilation time. The basic idea is to collect the execution time of via a profile run, then compile the application with the data from the profile run. First, we introduce another system call to increase the logical time of a task:

- __det_tick: This system call comes with a parameter of an integer. When it is called, the logical time will be increased by value defined by the parameter.

This system call should be inserted in the program where the logical time needs to be increased. In order to automate this instrumentation process, based on LLVM [22], we implemented two compiler passes to do the profiling and instrumentation.

Profile Pass In order to get the execution time of a program, we make a profile pass to collect the execution time at the granularity of basic block. During the compilation time, this compiler pass will assign a unique number to each basic block, and inserts time profiling functions around every basic block beyond a certain threshold in terms of number of instructions. Figure 3.6 shows a basic block instrumented with the profile functions in LLVM-IR. In this basic block, bbprof_start (line 3) and bbprof_end (line 16) are inserted at the beginning and the end of this basic block.

The profile run is launched by our profile launcher, which will keep track of the execution time of the application, and compute the average execution time for each instrumented basic block upon the application exits. In the end, all the gathered information will be output to a file for future use.

Logical Time Pass After the program finished one profile run with the instrumentation of profile pass, we can launch our compiler again to generate the final executable. The logical time pass will take the profile data file as input. This time at the end of each basic block, a __det_tick will be inserted with the parameter of a scaled execution time of the current basic block. So that the logical time will be bumped at the end of each basic block according to the actual execution time of each basic block. Fi gure 3.7 shows an example of instrumented basic block in LLVM-IR. This is the same basic block as we showed in Figure 3.6. In this example, Line 9 is the end of the basic block, it comes with a __det_tick system call with a value 2895535, which is generated and normalized from a previous profile run. In this basic block, line 5 is the most time consuming part in the entire program (pbzip2), as a result this basic block needs a relatively large tick increment.

3.2.2 Tick Bumping for External Events

When a task is sleeping in the kernel, usually it is in a system call and waiting for some events to wake it up. Especially for system calls like epoll_wait, poll and accept and other I/O system calls, the arrival time of the event is non-deterministic, as a result, we cannot simply use __det_tick to increase the logical time with a predefined value from a profile run, because we have no idea how long the thread will be sleeping in the kernel.

Some deterministic systems simply remove the sleeping tasks out of the deterministic schedule and put them back after they are back to user space. This is not applicable in a replication system like ours, as previously stated, the wake up time of those system calls might be different from the primary and secondary replica. As a result we must not abandon those sleeping tasks, and have to maintain the consistent state of the logical time for those tasks.

In order to let the token passing keep going with those blocking system calls, we need a way to keep bumping those thread's logical time while they are sleeping, a "Tick Shepherd" is implemented to dynamically bump the logical time of the threads that are sleeping in

```
if.end.23:                                        ; preds = %for.end
  %38 = load i8*, i8** %CompressedData, align 8
  %39 = call i32 (i32, ...) @bbprof_start(i32 249)
  %40 = load %struct.outBuff*, %struct.outBuff** %fileData, align 8
  %buf = getelementptr inbounds %struct.outBuff, %struct.outBuff* %40,
      i32 0, i32 0
  %41 = load i8*, i8** %buf, align 8
  %42 = load %struct.outBuff*, %struct.outBuff** %fileData, align 8
  %bufSize24 = getelementptr inbounds %struct.outBuff, %struct.outBuff*
       %42, i32 0, i32 1
  %43 = load i32, i32* %bufSize24, align 4
  %44 = load i32, i32* @_ZL12BWTblockSize, align 4
  %45 = load i32, i32* @_ZL9Verbosity, align 4
  %call25 = call i32 @BZ2_bzBuffToBuffCompress(i8* %38, i32* %outSize,
      i8* %41, i32 %43, i32 %44, i32 %45, i32 30)
  store i32 %call25, i32* %ret, align 4
  %46 = load i32, i32* %ret, align 4
  %cmp26 = icmp ne i32 %46, 0
  %47 = call i32 (i32, ...) @bbprof_end(i32 249)
  br i1 %cmp26, label %if.then.27, label %if.end.29
```

Figure 3.6: An instrumented basic block in pbzip2 with execution time profiling functions.

```
  (.....)
  %bufSize24 = getelementptr inbounds %struct.outBuff, %struct.outBuff*
       %35, i32 0, i32 1
  %36 = load i32, i32* %bufSize24, align 4
  %37 = load i32, i32* @_ZL12BWTblockSize, align 4
  %38 = load i32, i32* @_ZL9Verbosity, align 4
  %call25 = call i32 @BZ2_bzBuffToBuffCompress(i8* %32, i32* %outSize,
      i8* %34, i32 %36, i32 %37, i32 %38, i32 30)
  store i32 %call25, i32* %ret, align 4
  %39 = load i32, i32* %ret, align 4
  %cmp26 = icmp ne i32 %39, 0
  %40 = call i32 (...) @syscall(i32 321, i64 2895535)
  br i1 %cmp26, label %if.then.27, label %if.end.29
```

Figure 3.7: An instrumented basic block in pbzip2 with dettick.

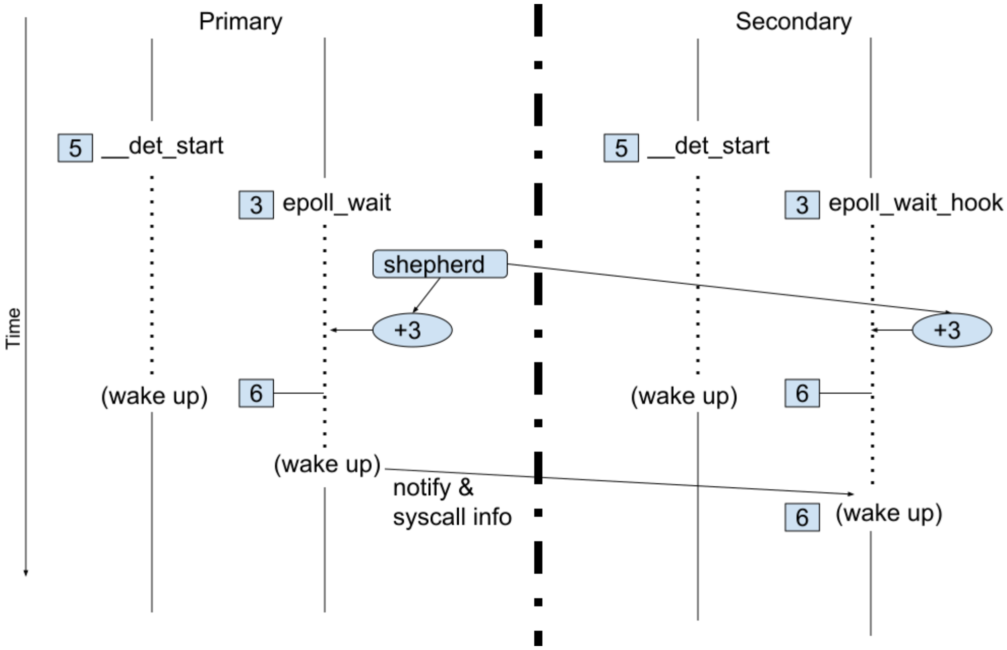

Figure 3.8: An example of tick bumping

such system calls. The Tick Shepherd is a kernel thread which is mostly sleeping in the background, whenever the token is passed on to a thread that is sleeping on external events or a thread is going to sleep with the token, the shepherd will be woken up to increase the sleeping thread's logical time and send the increased value to the replica. In the meanwhile the corresponding system call on the replica will be blocked at the entry point, and bumps its logical time according to the information from the primary. Figure 3.9 shows the simplified version of Tick Shepherd, it only runs on the primary replica. The syscall on the secondary doesn't proceed until the primary returns from the syscall. In this way we can make sure that when both of the syscalls wake up from sleeping, all the replicas will end up with a consistent state, in terms of logical time. The Tick Shepherd will keep bumping sleeping tasks logical time until for a given period the state of all the tasks comes to a stable point, where nobody makes a single syscall. After that, it will go back to sleep again.

Figure 3.8 shows an example of how Tick Shepherd works in action. In this example, tick shepherd detects the token is on a thread sleeping in epoll_wait, so it bumps its tick by 3 and sends this info to the secondary so that the token can leave this thread. And after the primary returns from epoll_wait, it sends a message to the secondary, so that the corresponding thread can start to execute its epoll_wait and uses the output from the primary as its own output. In order to be efficient, we only let Tick Shepherd to bump the system calls that for sure will be called for deterministic times, the current implementation covers all the major I/O related system calls.

```
/*
 * Definitions:
 * ns: current popcorn namespace
 * ns->token: a pointer pointing to the task holds the token
 * current->ft_det_tick: logical time of the task
 * current->ft_pid: replicated task unique identifier
 */
while (!kthread_should_stop()) {
  if (ns->task_count == 0 ||
    ns->wait_count == 0) {
    sleep(); // Sleep until some task wakes it up
    continue;
  }
  token = ns->token;
  tick = token->task->ft_det_tick;
  udelay(20); // delay for a small duration
  token2 = ns->token;
  tick2 = token2->task->ft_det_tick;
  // Which means the token hasn't been changed during the delay,
  // It's time to bump the tick
  if (token == token2 && tick2 == tick) {
    if (!is_waiting_for_token(token->task) &&
      (is_concerned_syscall(token->task->current_syscall)) {
        if (ns->wait_count != 0 &&
          token->task->bumped == 0) {
            bump_task = token->task;
            id_syscall = token->task->id_syscall;
            bump = ns->last_tick + 1;
            previous_bump = token->task->ft_det_tick;
            token->task->ft_det_tick = ns->last_tick + 1;
            update_token(ns);
            send_bump(bump_task, id_syscall, previous_bump, bump);
            continue;
        }
      }
   }
}
```

Figure 3.9: Simplified implementation of Tick Shepherd

3.3 Related Work

Deterministic execution is the most intuitive way of implementing a state machine replication system. However most of the existing deterministic systems are not suitable for production environments as mentioned in previous discussions [23], they are either domain specified, or too slow, or need hardware support. In the following subsections we will discuss the problems in each category of deterministic execution, also the existing solutions for applying deterministic execution to replication.

3.3.1 Deterministic Language Extension

Clik++ [24] is an parallel extension to C++ which makes creating parallel program easier. This extension provides a property that can indicate threads to be executed in a serial way, so that the determinism can be ensured. Grace [25] is also a C++ extension that adds a fork-join parallel schema to C++, it enforces the determinism of the execution with its underlying language runtime. Both of them are very limited to a specific parallel programming model, and existing applications need to be rewritten to achieve determinism.

3.3.2 Software Deterministic Runtime

Weak Determinism

Weak Deterministic systems usually only target on making synchronization primitives to be deterministic. Kendo[18], Parrot[16] and Dthreads[17] are three typical weak deterministic systems, they provide runtime substitutions for pthread library. By making pthread synchronizations to be deterministic, any race-free pthread-based application can be executed in a deterministic way. They are easy to be applied onto existing applications. Our implementation falls in to this category and the basic algorithm derives from Kendo. In order to address the logical time imbalance problem, Kendo relies on hardware counters to keep track of the program's progress in runtime, given the fact that hardware counters could be non-deterministic[21], for our replication use, it might not be worth to put too much engineering efforts to make the performance counters on both kernels to be synchronized.

Strong Determinism

Strong Deterministic systems aims to make every shared memory access to happen in a deterministic order. Consequence [26] and dOS [27] both provide an OS layer to make shared memory access to be deterministic, which is applicable for all kinds of parallel programming models. However their overhead is too high due to massive trapping to shared memory accesses, with this kind of overhead (at least 1X) it is not practical for them to be used in production. DMP [15] based on dOS, introduces hardware transaction memory to accelerate the memory trapping process. In our replication use case, such strong determinism is not needed, as we only need need the output of replicated applications to be the same. The effort for enforcing strong determinism would put too much unnecessary overhead.

3.3.3 Architectural Determinism

In [28] and [29], the authors both proposed architectural solutions to ensure memory access determinism. The goal for such systems is to track all the memory access and does versioning on the memory operations. By doing deterministic submission to the memory hierarchy, they are able to ensure the determinism of the parallel execution. RCDC [30] proposes a software/hardware hybrid solution to provide a relaxed deterministic access to the shared memory regions. All are promising solutions to provide a transparent deterministic execution environment, but those designated hardware support cannot be easily satisfied on commodity hardware.

3.3.4 Deterministic System For Replication

Almost all the deterministic systems mentioned the use case for replication but few provides an actually solution. Theoretically, all the deterministic systems mentioned so far are able to be applied for replication, but only for applications that do not have any network communication. The major challenge for replicating concurrent network applications is the arrival time of the network events is non-deterministic and unpredictable. In order to make the replicas be consistent, the replicas have to process the requests on the same state. All the weak deterministic systems mentioned so far either did not mention network operations(Dthreads[17]) or simply skip the threads doing such operations (Kendo[18], Parrot[16]), leave them out of the deterministic scheduling.

Actually skipping the threads sleeping in network events is applicable with some workarounds, as long as the system can ensure that when those threads are back from sleeping, all the replicas can be in the same state. A solution is to delay the wakeup time of those threads a little bit until all the replicas reach the same state. We investigated the skipping strategy with Kendo's algorithm, a possible solution is to bump the logical time of the sleeping threads to a relatively high value, so that when they are back to the deterministic schedule, no running thread can have a higher logical time other then them. We modelled such strategy with a multi-threaded network server in TLA+ and proved the correctness of it. However, in practical, it is very hard to pre-determine such a future logical time for the unpredictable network events, furthermore, delaying the wakeup time of those threads will sure have impact on the performance. As a result, we chose to not to skip any socket operations and ended up with the current Tick Shepherd solution.

Several works showed the same idea that network operations should not be skipped, dOS [27] mentioned a use case for replicating a micro web server, which uses the SHIM layer to block the network requests until the all replicas reach the same state. This solution will harm the performance badly and requires modifications to the application. Crane [3] utilizes Parrot[16] as the underlying deterministic system but without skipping the network operations. On top of that, Crane uses Paxos to bridge the gap between non-deterministic socket requests and the deterministic system, which ensures that all the replicas can receive the requests in the same state. However, Crane does not show any performance scalability in their work, and with our experiment on Crane, their Paxos layer becomes very unstable with large number of network requests.

Chapter 4

Nigoki: Schedule Replication

In chapter 3 we described using a deterministic system to ensure the applications on the primary and secondary replica can have the same thread interleaving. The major advantage of the deterministic system is that we can minimize the communication between the replicas. However the downside is that we need to precisely adjust the logical time to maintain decent parallelism for multithreaded applications. We showed various solutions to balance the logical time because we need to keep the execution to be fast and deterministic. If all the burdens come from being deterministic, can we break the determinism once for all but still keep the replicas to be synchronized? The answer is yes.

In this chapter we present Nigoki[1]: Schedule Replication. In this replication mode, we break the determinism entirely and use messages to synchronize every single synchronization primitives between the primary and replica.

For an application that has massive number of synchronization primitives, this approach might introduce overheads from the communication. Any latency in the the messaging will cause the secondary to fall behind the primary. Fortunately, our system is for inter-kernel replication, and Popcorn Linux provides a messaging layer with relatively low latency (basically memcpy from one kernel to another). As a result having massive massages between replicas won't put too much overhead to the replication.

4.1 Execute-Log-Replay

Before we get into the detail of this algorithm, let us revisit some important properties that are provided by the deterministic system.

- Serialization of deterministic areas. (The code region between detstart and detend).

[1] Means Unit 2, in Japanese

- Same total order of getting into deterministic areas on primary and secondary.

The first property is guaranteed by the fact that the logical time will not change during the execution of a deterministic area, and the second property is guaranteed by increasing the logical time in a same way on both primary and replica. As long as these two properties are guaranteed, the thread interleaving on both primary and secondary are sure to be the same (also for tick bump). By following this paradigm, in our Schedule Replication mode, we guarantee these two properties with the following approaches:

- Serialize deterministic areas with a global mutex on both primary and secondary.
- Log the sequence of getting into deterministic areas on the primary and replay it on the secondary.

Here we still use __det_start and __det_end to wrap around a code section that needs to be synchronized with the replica. Figure 4.1 shows a simplified version of __det_start and __det_end in Schedule Replication. Every thread in the namespace maintains a sequence number Seq_{thread} and the entire namespace maintains a sequence number Seq_{global}. On the primary, __det_start simply locks the global mutex, __det_end unlocks the global mutex, sends a tuple of $< Seq_{thread}, Seq_{global}, ft\_pid >$ to the secondary and then increase the value of Seq_{global} and Seq_{thread}. On the secondary, __det_start blocks until it receives a $< Seq_{thread}, Seq_{global}, ft\_pid >$ tuple corresponds to its caller thread, then holds the global mutex, and __det_end increases Seq_{global} and Seq_{thread}, then release the global mutex.

Figure 4.2 shows an example of how Schedule Replication works in action. In this example, T1 on the primary reached __det_start first and acquired the global mutex, which blocked T2 from getting into its __det_startȦfter the primary reached __det_end the global mutex is released and T2 was able to proceed. On the secondary, both T1' and T2' got blocked on __det_start at the beginning, no matter which one reached its __det_start first. T1' was able to proceed after T1 on the primary reached __det_end and sent the notification to the secondary. T2' proceeded in the same way as T1' did. With this, the timing of calling mutex_lock on the primary and secondary are synchronized on the primary and secondary.

For each namespace on the secondary, we have a queue for logging the incoming schedule replication message from the primary. The Popcorn message handler for schedule replication message simply appends the message into the queue tail and __det_start waits on the queue head to become the schedule sequence that it needs. A crucial prerequisite for this mechanism is that the message in the queue shall preserve strict FIFO sequence. Otherwise an out-of-order message in the queue tail will cause a deadlock in the system, because no __det_start will find the matching message in the queue tail. Our implementation guarantees the correct order of the messages, i.e, the messages are put in the queue in a monotonic sequence by their global sequence number Seq_{global}.

```
/*
 * Definitions:
 * ns: current popcorn namespace
 * ns->global_mutex: global_mutex in current namespace
 * ns->seq: global sequence number Seq_global
 * current->seq: task sequence number Seq_thread
 * current->ft_pid: replicated task unique identifier
 */
void __det_start()
{
    if (is_secondary(current))
        wait_for_sync(current->seq,
            ns->seq, current->ft_pid);
    lock(ns->global_mutex);
    current->ft_det_state = FT_DET_ACTIVE;
}
void __det_end()
{
    if (is_primary(current))
        send_sync(current->seq,
            ns->seq, current->ft_pid);
    current->seq++;
    ns->seq++;
    current->ft_det_state = FT_DET_INACTIVE;
    unlock(ns->global_mutex);
}
```

Figure 4.1: Simplified implementation of system calls for schedule replication

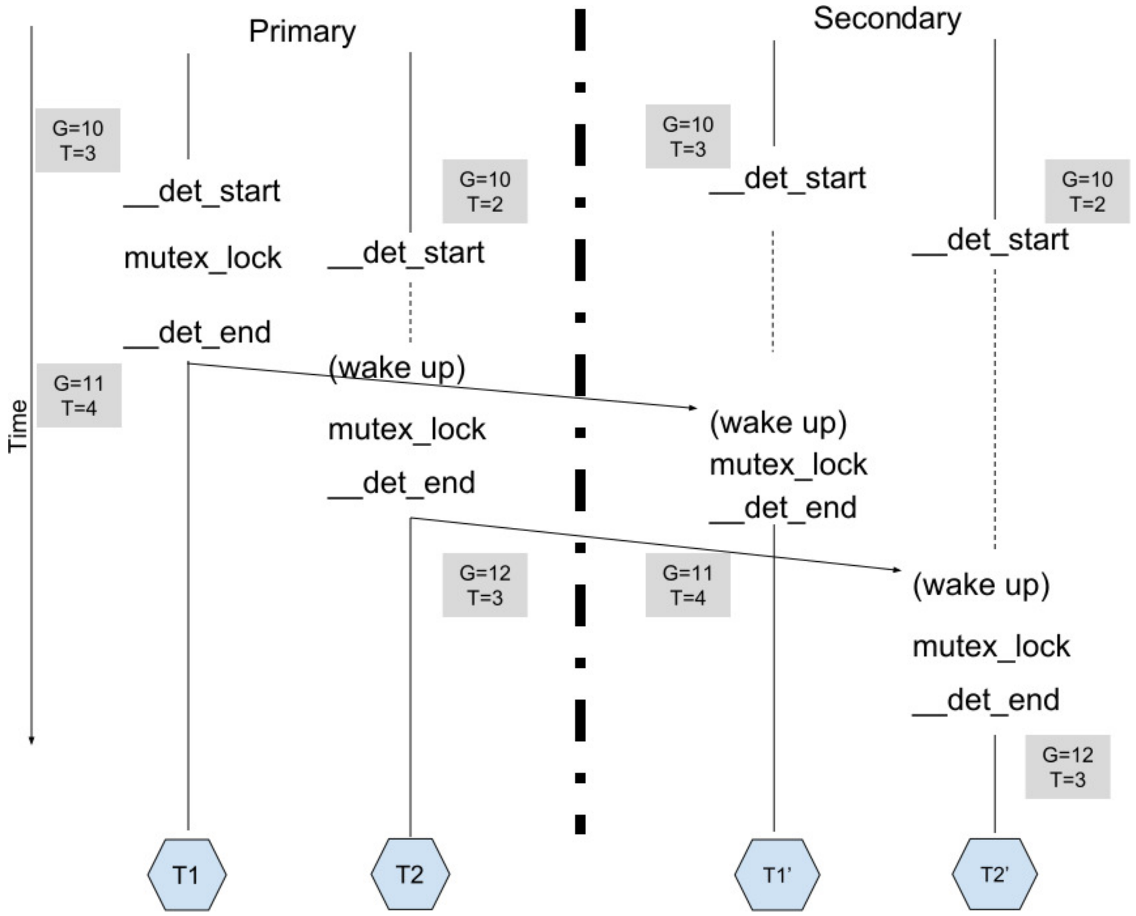

Figure 4.2: An example of Schedule Replication

- The synchronization message is sent with the global mutex on hold, this guarantees the monotonic sequence from the sender side.
- The messaging layer is strictly FIFO, which will not re-order the messages in its buffer, this guarantees the monotonic sequence from the receiver side.

4.1.1 Eliminate Deadlocks

As we mentioned in Section 3.1.1, wrapping all the lock acquisitions with __det_start and __det_end will occur the same deadlock issue, the reason is similar to the case in the Deterministic Execution because we don't release the lock acquisition order when the mutex is contended. The solution in Schedule Replication is similar to what we did in the Deterministic Execution, upon getting into sleep in futex, we release the global mutex and re-acquire it when it wakes up from futex. The futex modification mentioned in Section 3.1.1 is also applied in this case to ensure the determinism of waking up from futex.

4.2 Related Work

Several previous works also presented the idea of not using a deterministic system for replication. Midas [31] points out the non-determinism from both the thread-interleaving and system call outputs, and utilizes a compiler framework to trap those non-deterministic points. The primary records the output of trapped points and replay them on the replica. Rex [1] presents the same idea of logging the lock sequences on the primary and replay it on the replicas. It utilizes Paxos [32] to provide a consistent sequence of requests and locks across all the replicas. Comparing to our solution, the advantage of Rex is that it is able to use partial order lock synchronization to provide decent parallelism for different lock acquisitions. But the downside is that both of the solutions are not transparent enough. In Rex, applications need to be manually modified to adopt Rex. (300-500 lines of changes for each application, according to the evaluation part of the paper.) Moreover, both solutions cannot deal with non-determinism comes from an external library.

A more aggressive idea is not synchronizing the execution sequence at all. EVE [2], Recspec [8] and Zyzzyva [33] present the idea of running the replica speculatively without synchronizing the thread-interleaving. All of them assume for most of time the replicas are able to produces the consistent output, but whenever the replica diverges, the diverged one is forced to roll-back to a previous consistent state. Compare to our solution, EVE is implemented in JAVA and needs a good amount of manually annotation work to the applications, Zyzzyva seems to be highly customized and did not show evaluation for real world applications. For Recspec, it runs the application and the replicated one inside the same OS, which does not meet the requirement of our use case.

Chapter 5

Additional Runtime Support

With the implementation of the thread synchronization interface, we are able to control the thread interleaving for all the regions surrounded by __det_start and __det_end. In this chapter we will discuss the additional runtime support which eliminates some other non-deterministic facts that cannot be simply solved by __det_start and __det_end, and some optimizations to the current runtime. This chapter is organized as follows:

- Section 5.1 shows the non-deterministic facts come from some system calls and our system call synchronization mechanism.
- Section 5.2 shows how we instrument pthread primitives with _det_start and __det_end transparently.
- Section 5.3 shows how we create a consistent stdin, stdio and stderr interface for the replicated process.
- Section 5.4 shows an optimization for our system that allows relaxed determinism.

5.1 System Call Synchronization

During the execution of an application, for most of the system calls, given the same external input, the application on both primary and secondary can produce the same result, however there are still some system calls that are intrinsically non-deterministic, which will lead to divergence of the execution on all the replicas. As a result we have to synchronize the output of them to ensure the consistent final output of the applications on both sides.

Disabling vDSO vDSO(virtual dynamic shared object) is a mechanism that allows a system call to be done in user space, instead of having context switch to the kernel space.

This is done by having a shared memory section between the user space and the kernel. When the system call is initiated, the corresponding function in the vDSO library is called instead of trapping into the kernel, then the library will fetch the result from this shared memory area and return. This boosts the performance for some ”read only” system calls (like gettimeofday/time). However, in our case, if the system call does not go into the kernel space, we cannot track and synchronize them. Also, in order to synchronize the system call data we have to get into the kernel space anyway to send inter-kernel messages. So vDSO in our context becomes a burden to the implementation. As a result in our system we have to disable vDSO.

In Popcorn Linux, socket read/write/accept/close are already synchronized via the replicated network stack, here we implemented some other system calls that are strongly related to I/O results: gettimeofday/time, poll, epoll_wait. We did not implement select because it is relatively out-dated, modern network applications hardly use it. In the following subsections we will describe each synchronized system call in detail.

5.1.1 gettimeofday/time

gettimeofday and time are used for getting the current timestamp. Since the primary and secondary can not always have the same execution progress, the timing of calling gettimeofday/time might be different. For those applications that the output is time related, those system calls will cause output divergence. For gettimeofday/time, the primary simply copies the result to the secondary, when secondary executes the corresponding gettimeofday/time, it directly uses the output from the primary and bypasses its original path.

5.1.2 poll

poll is used for waiting on a set of file descriptors for I/O. A programmer can register a set of file descriptors to poll along with the type of events that is related to those file descriptors. poll takes an array of pollfd struct as shown in Figure 5.1. When it is called, it waits until one or more registered file descriptors become ready with registered events. When it returns, it fills the array with those file descriptors that are ready and returns the number of ready file descriptors. The user space application iterates the array and reacts to each file descriptor according to the events and revents field.

poll notification mechanism relies on the Linux VFS subsystem. However, as described in previous chapter, on the secondary kernel the replicated TCP/IP stack will bypass the original execution path for accept/read/write on sockets, in other words, the VFS subsystem is partially bypassed. As a result, poll will not be woken up properly on the secondary even when the event already arrives, which leads to a different output other than the primary.

The solution is similar to time/gettimeofday, we simply send the output of poll to the

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
    int   fd;          /* file descriptor */
    short events;      /* requested events */
    short revents;     /* returned events */
};
```

Figure 5.1: poll prototype and pollfd data structure

secondary. As shown in Figure 5.1, the output of poll is the fds array and the return value. Upon receives the information, the secondary uses this as the output of itself and bypasses its original execution path.

5.1.3 epoll_wait

Similar to poll, epoll_wait is also used for waiting on a set of file descriptors for I/O. It waits on a set of registered file descriptors and outputs the ready ones to an epoll_event array. Due to the implementation of our replicated network stack, epoll mechanism has the same problem as poll. Figure 5.2 shows the prototype of epoll_wait and epoll_event structure. Compare to the relatively simpe pollfd structure, epoll_event contains a data field which can be an arbitrary data structure. It is OK to just copy the data field to the other side if it only contains integers. However if this field is a pointer, due to the non-determinism of memory address on both side, simply passing the pointer to the other side may lead to an illegal memory access. As a result, on the secondary, along the output path of epoll_wait, we need to find the corresponding data structure in its own address space.

On the primary kernel, once the epoll_wait is ready to return, it will send a message which contains the current epfd, all the ready file descriptors and the value of events field of every file descriptor. Upon the secondary receives the message, it will search the RB tree associated to the given epfd, find the previous registered epoll_event of the ready file descriptors, and overrides the events field with the information from the primary. At the end, return to the user space with the array of epoll_event and bypass the original epoll_wait execution.

5.2 Interposing at Pthread Library

In Chapter 3 and Chapter 4 we described how to wrap the pthread primitives with __det_start and __det_end to ensure the same thread interleaving for the replicated application on the primary and the secondary. Manually instrument the code is tedious, one has to find every single pthread primitive in the code. Moreover, if an application uses an external library

```
int epoll_wait(int epfd, struct epoll_event *events,
                     int maxevents, int timeout);

typedef union epoll_data {
    void    *ptr;
    int      fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;

struct epoll_event {
    uint32_t     events;    /* Epoll events */
    epoll_data_t data;      /* User data variable */
};
```

Figure 5.2: epoll_wait prototype and epoll_event data structure

that uses pthread, it will be even more troublesome to recompile the needed external library. An intuitive solution is to modify the pthread library and wrap our __det_start and __det_end directly in the pthread code. However updating the glibc of a system can be very dangerous and might harm other applications that don't need to be replicated. Fortunately we can use LD_PRELOAD linker trick to implement a clean solution.

LD_PRELOAD In Linux, the behaviour of the dynamic linker can be altered by setting LD_PRELOAD environment variable. This can change the runtime linking process and make the linker to search for symbols in the path defined in LD_PRELOAD. With this trick we are able to alter the behaviour of glibc without actually changing it. We implemented our LD_PRELOAD library with instrumented pthread functions in it, and the namespace launching script will automatically set LD_PRELOAD environment variable to be the path of our library, so that only the application running in the namespace will be affected by our LD_PRELOAD library. In the upcoming sections we will describe how we wrap pthread functions in our LD_PRELOAD library.

5.2.1 Interposing at Lock Functions

Figure 5.3 shows the implementation of pthread_mutex_lock in our LD_PRELOAD library. Line 9 loads the real pthread_mutex_lock function from the real pthread library, in Line 12 we simply call this function with __det_start and __det_end wrapped around. In our LD_PRELOAD library, we wrapped all the pthread lock functions include pthread_mutex_lock, pthread_mutex_trylock, pthread_rwlock_rdlock, pthread_rwlock_tryrdlock, pthread_rwlock_wrlock, pthread_rwlock_trywrlock. There we need no special treatment for unlock primitives, as long

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
{
    int ret;
    static int (*pthread_mutex_lock_real)(pthread_mutex_t *mutex) = NULL;
    if (!handle) {
        handle = dlopen(PTHREAD_PATH, RTLD_LAZY);
    }
    if (!pthread_mutex_lock_real)
        pthread_mutex_lock_real = dlsym(handle, ”pthread_mutex_lock”);

    syscall(__NR_det_start);
    ret = pthread_mutex_lock_real(mutex);
    syscall(__NR_det_end);

    return ret;
}
```

Figure 5.3: pthread_mutex_lock in the LD_PRELOAD library

as the calling sequence of lock operations are determined, the return timing for the lock operations will follow the same sequence [19].

5.2.2 Interposing at Condition Variable Functions

Condition Variables are much more complicated than mutex locks. In the glibc implementation, it involves multiple internal lock and unlock operations. As a result simply wrapping pthread_cond_wait with __det_start and __det_end will not work, because of multiple non-deterministic execution points are inside the implementation. Figure 5.4 shows the brief flow of the pthread_cond_wait in glibc implementation, yellow blocks are the lock acquisitions. cond-¿lock is a lock inside the condition variable data structure, it is used to provide mutual exclusion for the futex value for the condition variable. futex_wait will wait until cond¿futex differs from futex_val. When it wakes up, it will check again if this condition variable is contended, if so, go back to futex_wait again. If not, re-acquire the mutex lock and return. Every single lock acquisition here is a non-deterministic point, which leads to passing different values to futex_wait on primary and replica, which in turn leads to diverged wakeup timing of pthread_cond_wait.

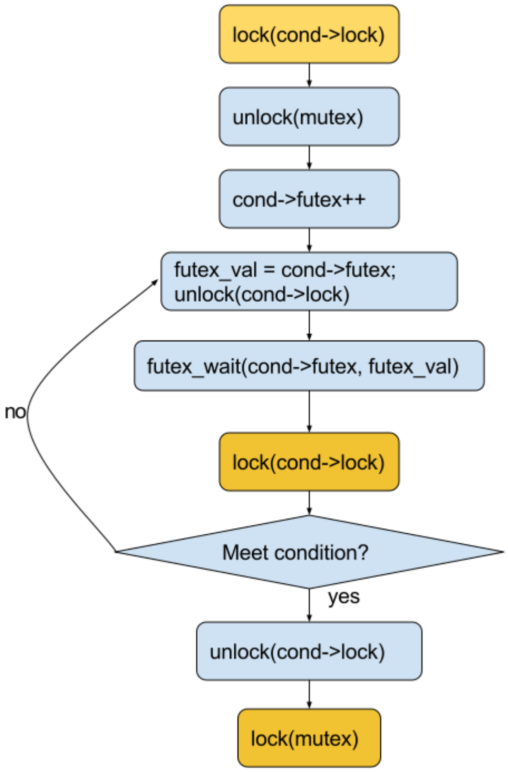

Figure 5.4: glibc pthread_cond_wait internal work flow

In our LD_PRELOAD library, we re-implemented pthread_cond_wait following the existing glibcś implementation, and wrapped every lock acquisition with __det_start and __det_end, we also did the same wrapping for pthread_cond_signal. With this, we are able to make sure that the pthread_cond_wait can return at the same timing with the same condition variable on both primary and secondary.

5.3 stdin, stdio and stderr

In the booting process of Linux, init is the very first userspace process and it creates the file descriptors for stdin, stdio and stderr. All upcoming process inherit those three file descriptors from init. This gives all the processes the ability to interact with a terminal device, also gives the fact that 0, 1 and 2 are the ”reserved” file descriptor numbers in a process, any newly created file descriptor starts from 3. However, as we described in previous, Popcorn Linux generates a replicated process on the secondary from the kernel space, pretty much like how init is created. As a result the replicated process doesn't inherit the stdin, stdio and stderr and newly created file descriptor starts from 0. This creates divergence on applications which take file descriptor numbers as some sort of input. An example is poll and epoll_wait, since we copy the ready file descriptors on the primary to the secondary, the divergence on file descriptor numbers will lead to unexpected results for upcoming I/O operations after poll or epoll_wait. The solution is very straightforward, upon the creation of the replicated process on the secondary kernel, we look for an available pts device, and use it as the terminal for stdin, stdio and stderr of the replicated process. In this way we are able to have consistent file descriptor numbers on primary and replica, and also be able to see the replicated processś console output.

5.4 Synchronization Elision

In some applications, not all the lock acquisition must be synchronized. For example, the lock primitives in a memory allocator don't affect the final output at all, as a result we can relax the determinism for those locks. In both synchronization strategies, we multiplexes __det_tick with tick number 0 as the hint for relaxing the determinism of the next __det_start when the that __det_start is called, the system call does nothing and simply returns. In this way we are able to boost the performance of some applications with manually instrumentation.

Chapter 6

Evaluation

In this chapter we will show some experiment results of our system. We will use various applications which will cover all the aspects of our implementation includes thread interleaving synchronization, application instrumentation and system call synchronization. With all the evaluation, we will answer the following questions:

- Correctness: Given the same input, can the primary and secondary consistently generate the same output?
- Performance: Compare to non-replicated execution, how much overhead is introduced by our system?
- Overhead: What do we need to pay for doing replication?

Evaluation Setup All experiments were run on a server machine with 4 AMD Opteron 6376 Processors (16 cores each, 2.3Ghz), which is 64 cores in total. The total RAM is 128GB. Our Popcorn Linux kernel was installed on Ubuntu 12.04 LTS. We partitioned the hardware resources into half, one for the primary and one for the secondary. Each of them has the full control of their own 32 cores and 64GB RAM. The machine comes with a 1Gbps high speed connection. For benchmarking server applications, we used another server machine in the same rack, connected to the same switch, to act as the benchmark client.

6.1 Racey

We used a variant of racey [34] to evaluate the correctness of our system. racey benchmark is a set of concurrent programs which read and write some shared data concurrently with various concurrent models. With a non-deterministic system, all the benchmark will create a different result during each different run. We use racey to validate if we can have the same thread interleaving on primary and secondary, which should lead the same output on both primary and secondary.

racey-guarded racey-guarded has a global array, it uses pthread to create multiple threads and modify the global array concurrently. The access to the global array is protected by pthread_mutex_lock. We tested this one without any modification to the application. With both synchronization algorithms, we are able to create consistent results on the primary and secondary for over 100 consecutive runs.

racey-forkmmap racey-forkmmap utilizes mmap to create a shared memory area, and uses fork to create multiple processes to read and modify the shared memory area. We manually added __det_start and __det_end around each access to the shared memory area. With both synchronization algorithms, we are able to create consistent results on the primary and secondary for over 100 consecutive runs.

racey-tcp Based on the idea of racey, we developed racey-tcp to stress the determinism for network I/O related tasks. racey-tcp uses pthread to create multiple threads. One thread listens to the socket, whenever a new connection arrives, it puts the connection into a queue, other threads retrieve the connection from the queue, read the data on that connection and write the data into a file. For this benchmark, we wrapped the write system call for writing to the file with __det_start and __det_end. With both synchronization algorithms, we are able to create consistent output file on the primary and secondary for over 2000 requests.

6.2 PBZip2

PBZip2 [35] is the parallel version of bzip2. The concurrent model of this application is a typical producer-consumer model, as shown in Figure 6.1. The FileReader thread reads the content of the file, break the input data into data chunks and put all the chunks into a queue. Worker threads get the data chunks from the queue and do the compression/decompression, after all put the produced data to another queue. The FileWriter will keep getting products from the queue and write them to the final zip file. Multiple pthread_mutex_lock and pthread_cond_wait functions are applied to provide the mutual exclusion to the access of the queues.

For PBZip2, the time consuming part is the place where it calls the libz2 compression/decompression functions. In this benchmark, we utilized the execution time profiling instrumentation to balance the logical time for the deterministic execution, while for schedule replication nothing is modified. The benchmark is to compress a 177MB file with different

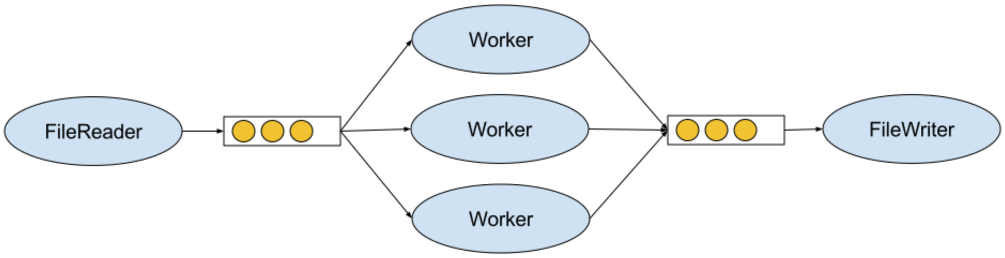

Figure 6.1: pbzip2 concurrent model

System Call	Use in the Application
gettimeofday	Calculate execution time/pthread_cond_timed_wait

Table 6.1: Tracked system calls used by pbzip2

thread counts, here we measure the performance with the total execution time reported by pbzip2. Other than that, no manually code modification was applied to the original source code.

Table 6.1 shows the system calls that are used by pbzip2, we only show the system calls that are tracked and synchronized by our system. In pbzip2, gettimeofday is used for showing the time spent on the whole process, which it is not critical to the output of the application. However pthread_cond_timed_wait also uses gettimeofday to calculate the timeout for the wait time, which is critical to the consistency of the execution.

Correctness For Deterministic Execution, any mismatch of the schedule will lead to different calling sequence of gettimeofday on the primary and secondary, which will result different reported execution time. For Schedule Replication, any mismatch of the schedule will lead the secondary waiting for a wrong schedule event forever. Neither of the case happened during the benchmark, the correctness of the replication thus proven.

6.2.1 Results

Figure 6.2 shows the execution time of vanilla Linux, Deterministic Execution and Schedule Replication. Both replication modes achieved decent scalability. However, as we can see in Table 6.2, both algorithms' overhead increases with the thread count. One important overhead source for both replication modes comes from the serialization of all the synchronization primitives. With increasing thread count, the downside of breaking the parallelism of accessing those regions becomes more obvious.

For Deterministic Execution, another type of overhead comes from the logical time imbalance. As we mentioned before, the execution profiler takes the average execution time of

Thread count	Deterministic Execution	Schedule Replication
2	14.27%	0.89%
4	29.47%	4.08%
8	44.77%	6.72%
16	54.44%	24.7%
24	63.39%	36.3%

Table 6.2: pbzip2 Overall Overhead of Each Replication Mode

basicblocks. However the execution time may vary during the actual run, especially for those basic blocks with file I/O, which has non-determinisitc execution time. Although the instrumented pbzip2 showed decent scalbility, the performance still could have been better if the logical time incremental could be more precise.

6.2.2 Message Breakdown

Figure 6.3 shows the number of messages that were used during the benchmark. In all the figures, "Schedule Messages" means the messages for Tick Shepherd in Deterministic Execution, while in Schedule Replication this stands for the messages for logging the execution sequence. This result is expected as we assume that Deterministic Execution doesn't require too much communication between the replicas. Here all the system call messages are for gettimeofday, and most of them are from pthread_cond_timed_wait. In pbzip2, pthread_cond_timed_wait is used by Worker threads to wait for available data chunks. An interesting finding is that for the same benchmark Deterministic Execution invoked less system calls than Schedule Replication. This is because for Deterministic Execution there is a minor logical time imbalance issue, where the FileReader and FilerWriter had more chance to run compare to Worker threads. In this situation, whenever a Worker thread has chance to run, it is very likely to find an available data chunk in the queue, thus no need to call pthread_cond_timed_wait. However for Schedule Replication, the arbitrary thread interleaving causes the Worker threads having more chance waiting on an empty queue, which leads to more invocation to pthread_cond_timed_wait.

6.3 Mongoose Webserver

Mongoose [36] is a compact multithreaded webserver. The concurrent model is shown in Figure 6.4. The MasterThread opens a listening socket, uses poll to wait for the incoming connections on the listening socket. Whenever a connection comes, the MasterThread accepts it and put the file descriptor to a queue. WorkerThreads get the connections from the queue and make the response to the clients. Table 6.3 shows the system calls that are

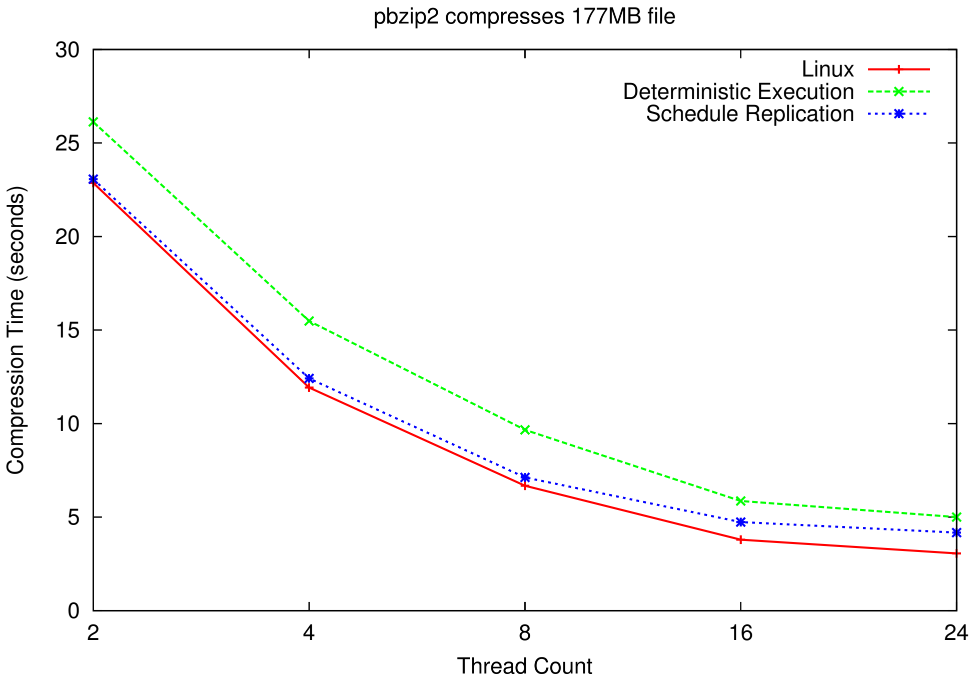

Figure 6.2: pbzip2 performance

System Call	Use in the Application
time	Generate HTTP header
poll	Wait for accept, read and write

Table 6.3: Tracked system calls used by mongoose

used by mongoose. The non-deterministic points in mongoose comes from both the thread-interleaving and system call output: diverged thread-interleaving leads to WorkerThreads handling incorrect sockets; diverged system call output leads to incorrect socket state and output value.

We used ApacheBench to stress test mongoose with different file sizes and different mongoose thread counts. For each benchmark set, we used 100 concurrent connections to make 20000 requests in total. For our benchmark on Deterministic Execution, since the file I/O is set to be blocking in mongoose and we do not track file I/O with Tick Shepherd, we manaully added a __det_tick right before the file I/O system call with an optimal value (only 1 line). Other than that, nothing was changed for mongoose.

Correctness For both replication mode, any mismatch will lead to either different thread handling different socket, or divergence in the HTTP responses. Neither of the case happened

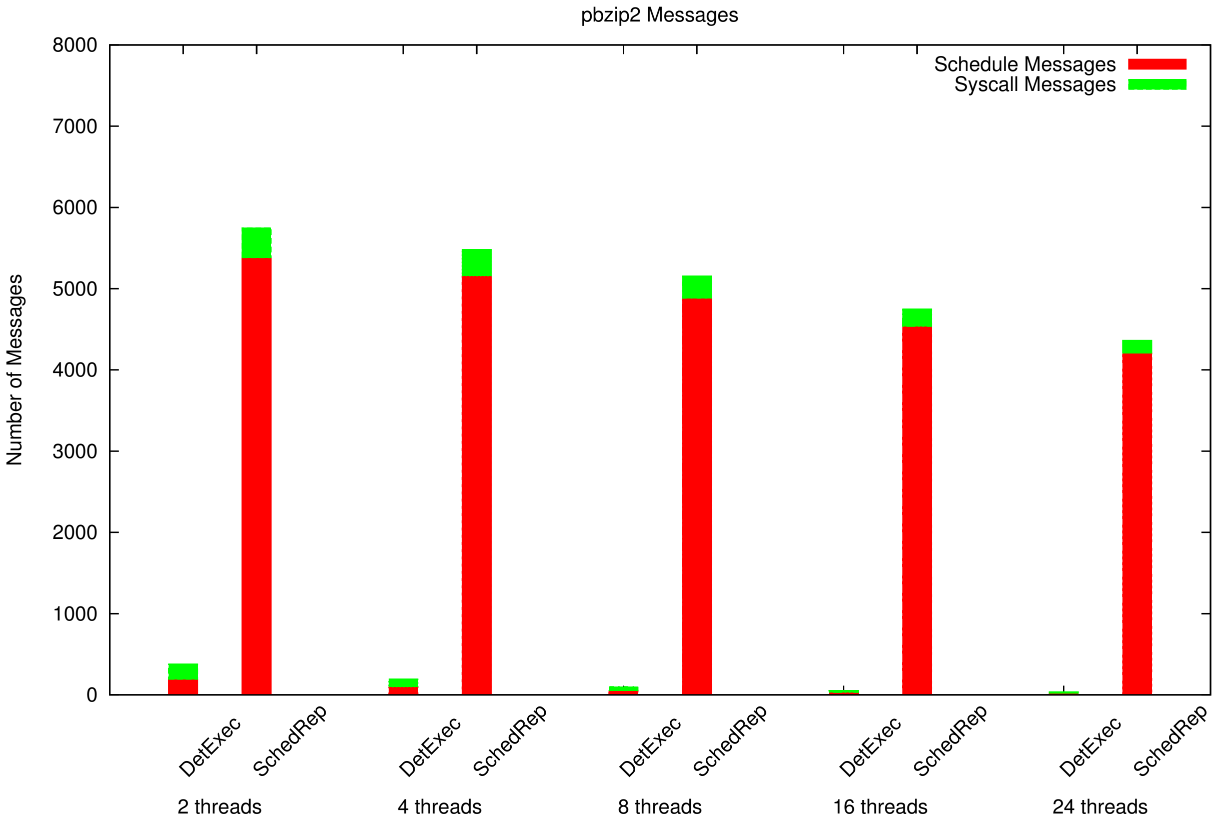

Figure 6.3: pbzip2 messages

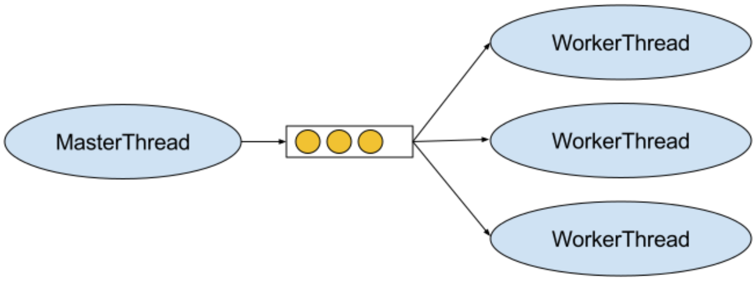

Figure 6.4: mongoose concurrent model

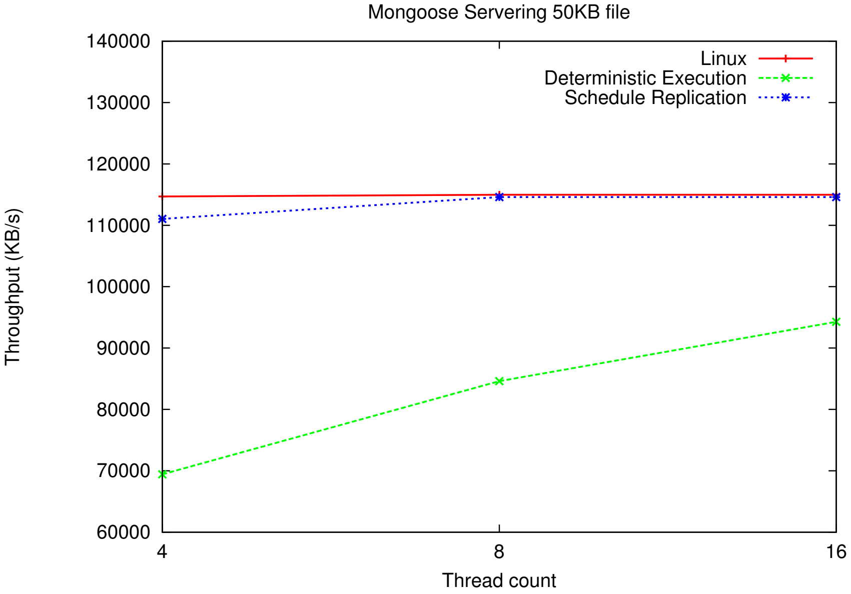

Figure 6.5: mongoose performance for 50KB file requests

Thread Count	Deterministic Execution	Schedule Replication
4	25.22%	1.35%
8	15.72%	0.27%
16	9.82%	0.23%

Table 6.4: Mongoose Overall Overhead of Each Replication Mode

during the benchmark, the correctness of the replication thus proven.

6.3.1 Results

Figure 6.5, 6.6 and 6.7 show the performance under different workload, and Table 6.4 shows the overall overhead of each replication mode. Deterministic Execution's overhead becomes lower as the thread count goes up, but still higher than Schedule Replication. This is due to blocking socket write and file I/O operations. Because the time spent inside those blocking I/O varies all the time, our manually inserted __det_tick couldn't precisely increase the logical time for every call, as a result we had to suffer the performance loss from logical time imbalance. In the meanwhile, Schedule Replication showed near zero overhead compare to the baseline.

Unlike pbzip where the overhead becomes higher when the thread count increases, mongoose's overhead decreases with more number of threads. This is because monoogse only has

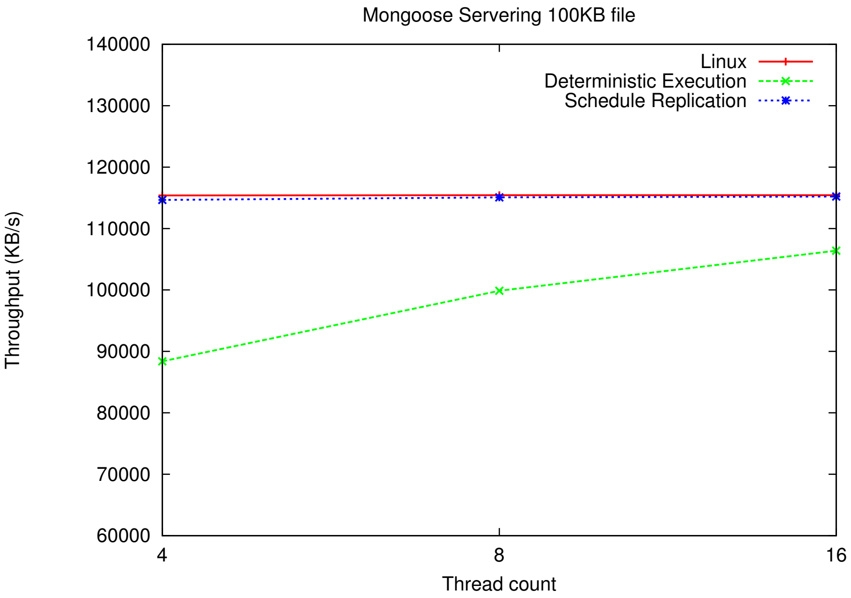

Figure 6.6: mongoose performance for 100KB file requests

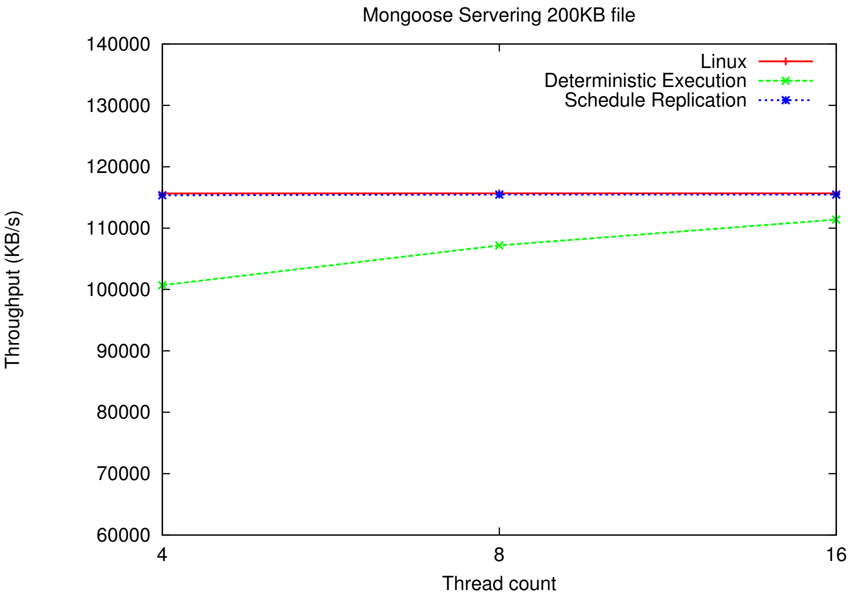

Figure 6.7: mongoose performance for 200KB file requests

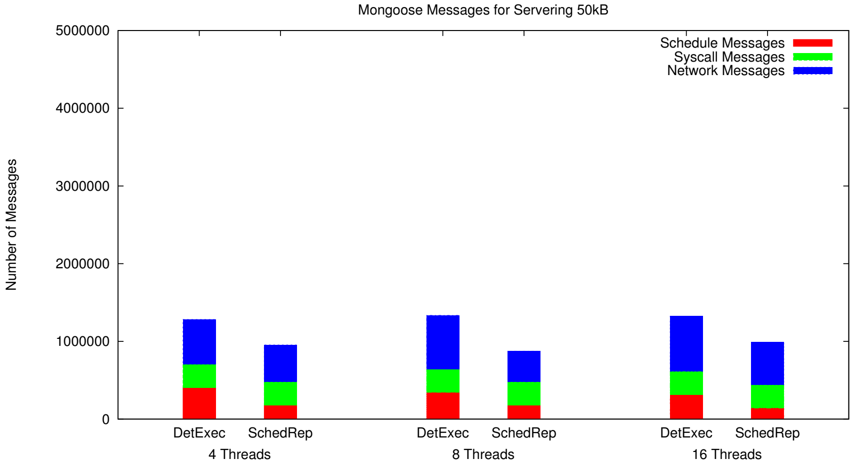

Figure 6.8: mongoose messages for 50KB file requests

two condition variables and one mutex lock, the side effect of serializating pthread primitives is minimal and the performance thus can scale.

6.3.2 Message Breakdown

Figure 6.8, Figure 6.9 and Figure 6.10 show the breakdown of overall messages for each benchmark set. The notation is the same as the previous section. Where ”Schedule Messages” means the messages for Tick Shepherd in Deterministic Execution, a in Schedule Replication this stands for the messages for logging the execution sequence.

Actually, we need much more messages for deterministic execution, which contradicts the assumption we made for Deterministic Execution. In mongoose, socket write and file I/O are blocking operations, and bigger network payload leads to more socket calls, thus we need more messages for the Tick Shepherd to synchronize the tick bumps. However for Schedule Replication, since the messages for scheduling only depends on the number of synchronization primitives, which totally depends on the number of requests (not the size), as a result, across all the benchmarks, the number of schedule messages for Schedule Replication show a near constant value across all the benchmarks.

6.4 Nginx Webserver

Nginx [37] is a sophisticated webserver with multiple threading modes. In this benchmark we used the threadpool setup [38] for our benchmark. As shown in Figure 6.11, in this threading mode, the additional threads are only for doing file I/O operations. The MasterThread waits

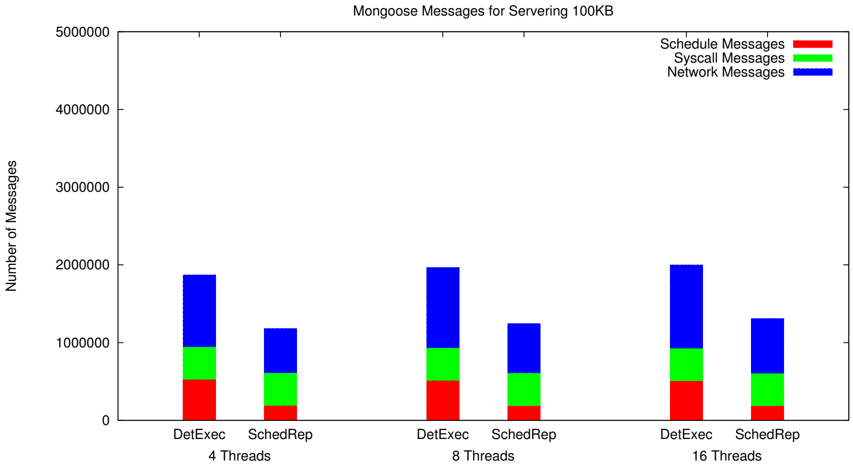

Figure 6.9: mongoose messages for 100KB file requests

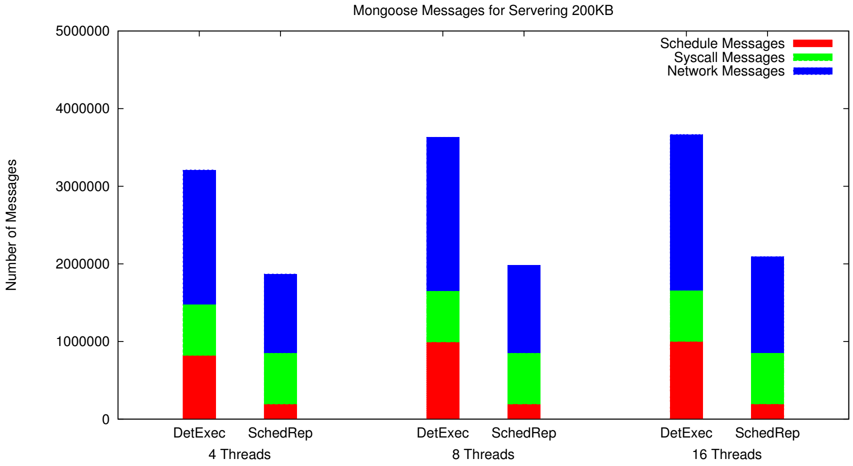

Figure 6.10: mongoose messages for 200KB file requests

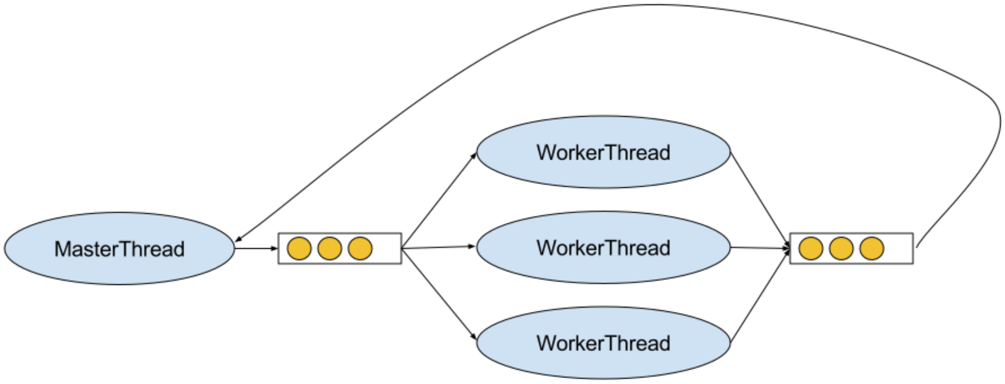

Figure 6.11: Nginx thread pool model

System Call	Use in the Application
gettimeofday	Generate HTTP header
epoll_wait	Wait for accept, read and write

Table 6.5: Tracked system calls used by nginx

on the listening socket, whenever a file request is coming, it hands over the request into a queue. WorkerThreads get notified whenever a new request is coming and try to retrieve one task from the queue. Whenever the content of the requested file is loaded into memory by a WorkerThread, a handle will be put into another queue and the MasterThread will get notified. In the end, the MasterThread will retrieve the handle and send the content of the file back to the client.

Table 6.5 shows all the tracked system calls used by Nginx. All of the I/O operations are asynchronous and heavily relies on epoll mechanism.

The same as mongoose, we used ApacheBench to stress test the server with different file sizes and different mongoose thread counts. For each benchmark set, we used 100 concurrent connections to make 20000 requests in total.

Nginx has multiple explicit atomic operations that might lead to non-deterministic results, here we applied __det_start and __det_end around those operations to maintain the total order of those operations. Another non-deterministic part in Nginx is that it uses eventfd to communicate among threads. As we mentioned before, all inter-thread communication need to be deterministic on all the replicas, as a result, we applied __det_start and __det_end around the read and write operations of eventfd to synchronize the total order of those operations. Other than that, no further modification were applied. In total, we added around 60 LOC to the original code.

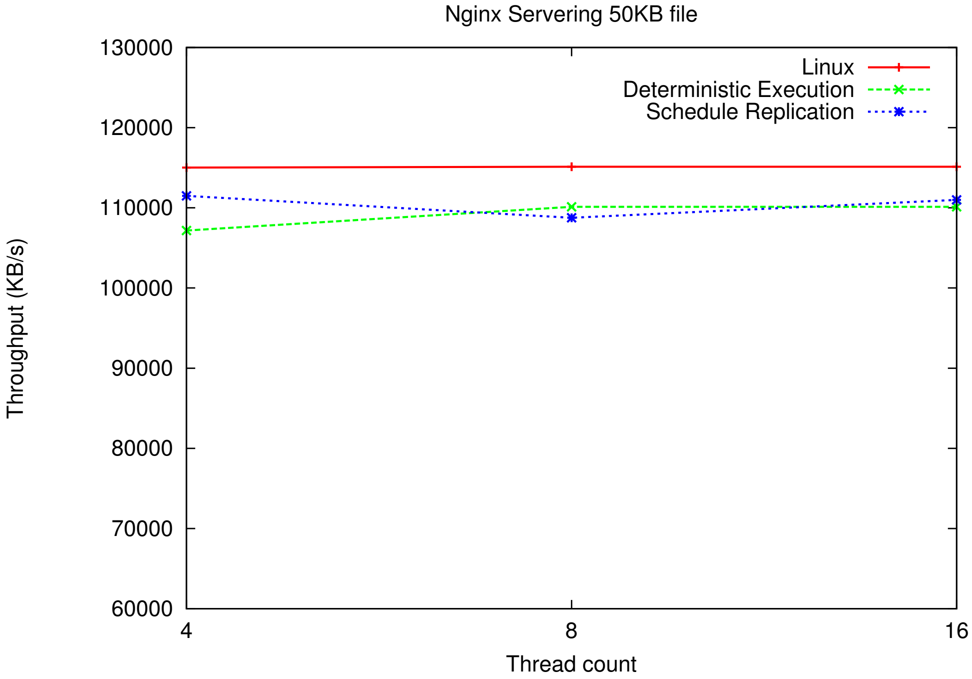

Figure 6.12: nginx performance for 50KB file requests

Thread Count	Deterministic Execution	Schedule Replication
4	3%	1.07%
8	1.62%	1.96%
16	1.6%	1.31%

Table 6.6: Nginx Overall Overhead of Each Replication Mode

Correctness Similar to mongoose, any mismatch will lead to either different thread handling different socket, or divergence in the HTTP responses. Neither of the case happened during the benchmark, the correctness of the replication thus proven.

6.4.1 Results

The reason we see the result is not scaling (even for Linux) is because this threading mode for Nginx is only for concurrent file I/O. During our benchmark, we've already achieved the best of our filesystem even with lower thread counts, although all the threads were loaded, putting more threads won't increase the performance too much. However, the purpose of the benchmark is to show that how will different thread counts affecting our system's overhead (comparing to baseline), a non-scaling result still makes sense here.

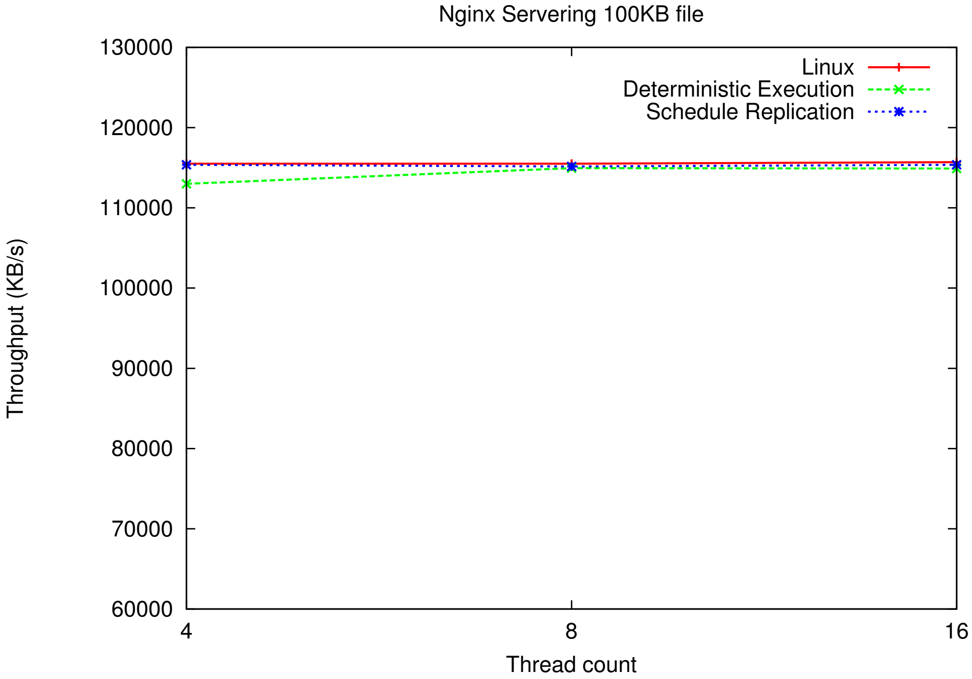

Figure 6.13: nginx performance for 100KB file requests

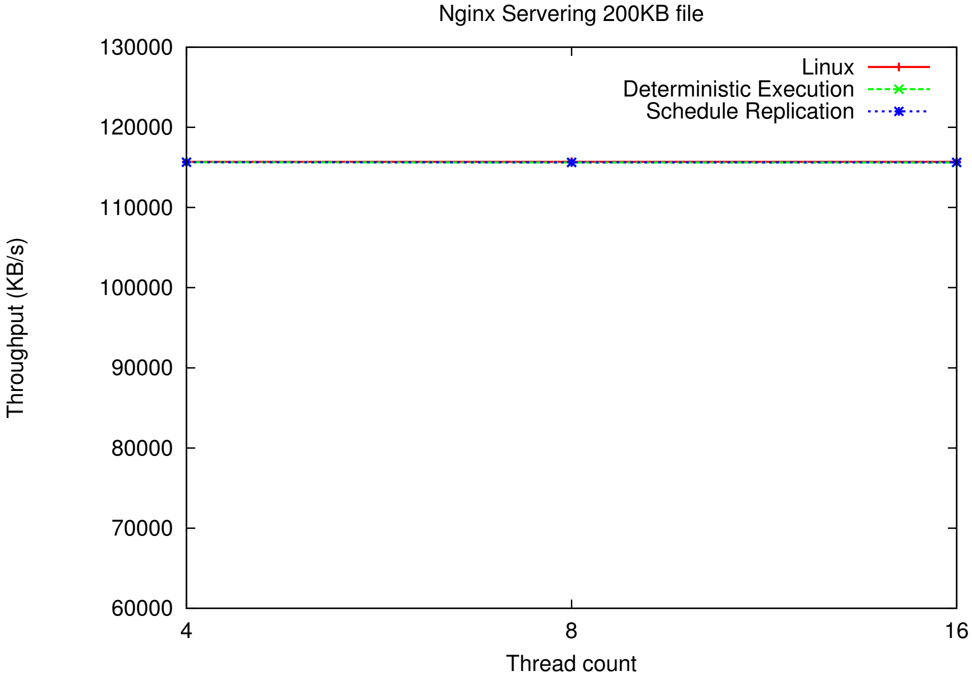

Figure 6.14: nginx performance for 200KB file requests

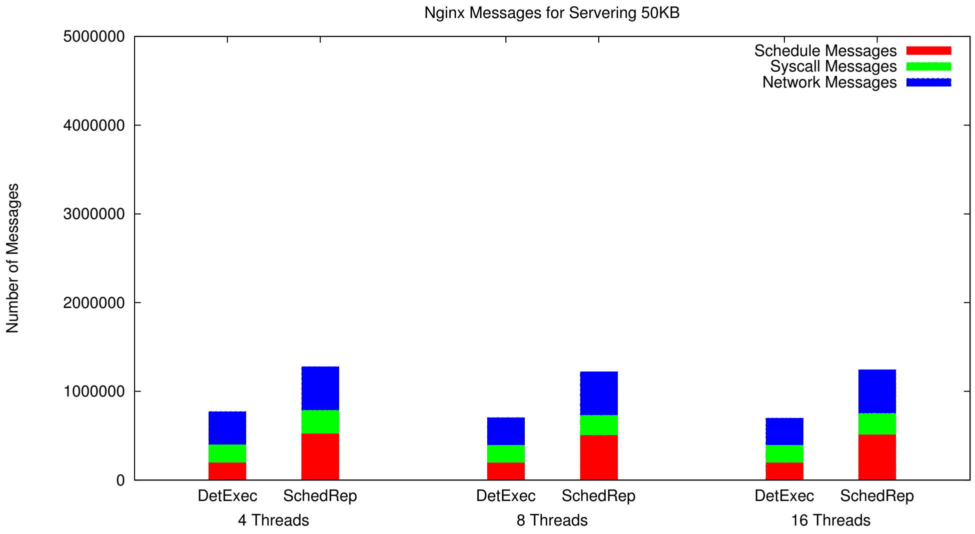

Figure 6.15: nginx messages for 50KB file requests

Unlike mongoose, the result of Nginx showed both replication modes can achieve very small overhead. When we take a step back to deterministic execution, the major performance hurdle is the logical time imbalance, which happens when there are some time consuming code sections. However, since nginx fully utilizes asynchronous file and network I/O, everything is super fast. In the context of deterministic execution, we can say in Nginx there is no significant time consuming part that will cause logical time imbalance. Therefore Deterministic Execution can perform its best.

6.4.2 Message Breakdown

Figure 6.12, figure 6.13 and figure 6.13 show the message break for our benchmark. As mention in the beginning of this section, Nginx needs more __det_start and __det_end other than pthread primitives, so Schedule Replication uses way more messages than Deterministic Execution. Another interesting fact is that since asynchronous I/O operations are really fast in Nginx, in Deterministic Execution, the Tick Shepherd barely made a tick bump, which is also a reason that Deterministic Execution could perform much better than it was in mongoose. While in mongoose, almost every blocking I/O had at least one tick bump.

6.5 Discussion

The selected benchmarks were used to stress different aspects of this project. Although there are still some optimizations to go, we still achieved decent overhead for different types

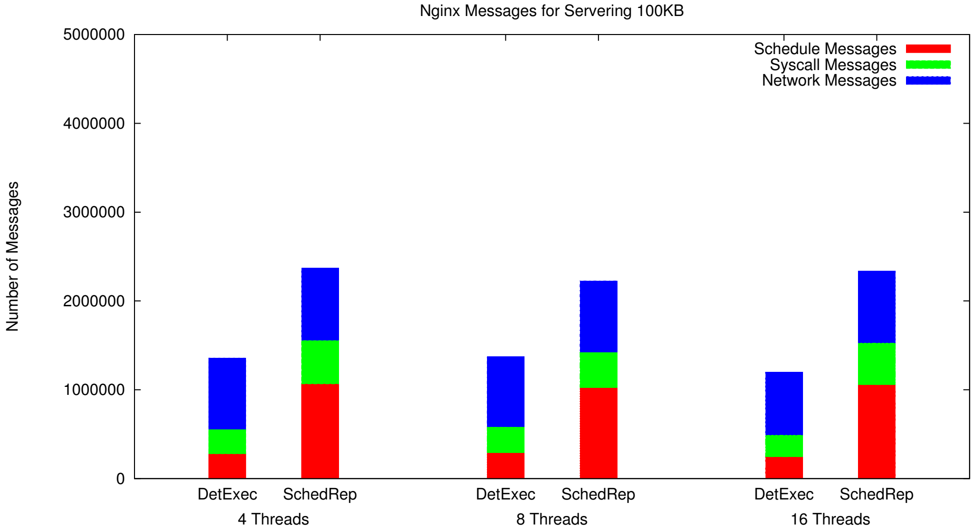

Figure 6.16: nginx messages for 100KB file requests

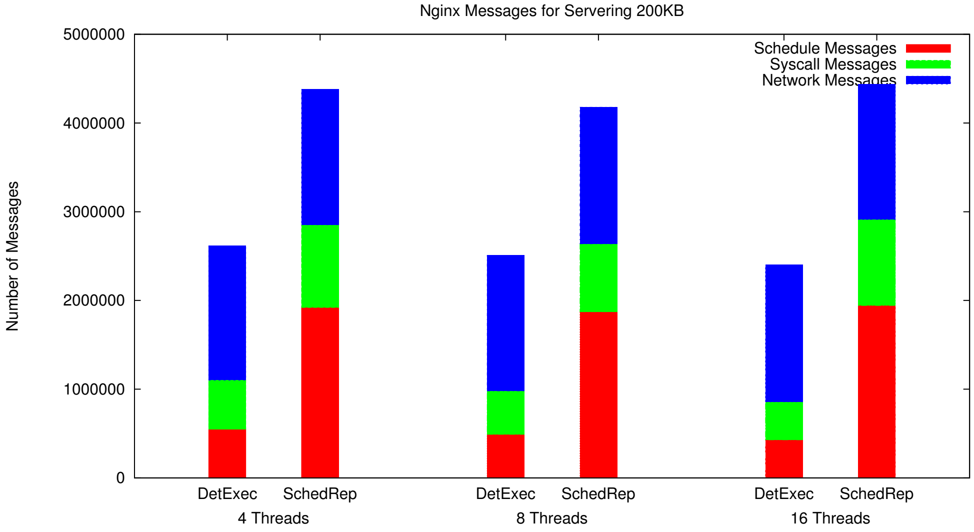

Figure 6.17: nginx messages for 200KB file requests

of concurrent models and different application types. Here we will discuss some interesting issues during the benchmark and some findings from the results.

6.5.1 Benchmark for Nginx's Multi-process Mode

So far for all the marco benchmarks, we only showed benchmarks for multi-threaded applications. However our system also supports replication for multi-process applications (as we showed for racey microbenchmark). Nginx supports multi-worker mode, which spawns multiple worker processes, all of the worker processes wait on the listening socket together. Nginx implements the accept_mutex [39], which is lock across processes, to ensure an incoming request only wakes up one waiting worker. In order to figure out whether our replication works for this concurrent model or not, we manually instrumented the accept_mutex acquisition with __det_start and __det_end . During the benchmark, both primary and secondary were able to end up with the same state all the time. However no matter how many workers we put, it was always only one worker handling all the request. This is due to a relatively low workload we used in our benchmark. However, because of the limitation of our network card (1Gbps), we couldn't apply heavier workload to saturate the worker. With Nginx's design, the accept_mutex mechanism will not pass the request to another worker if the current worker is not saturated. Which is the reason we only saw one worker handling all the requests.

Unlike the non-scaling result with the threadpool mode, where all the threads were fully loaded, the unbalanced workload in multi-worker mode seems not making any sense. As a result we didn't include the result in this chapter.

6.5.2 Deterministic Execution's Dilemma

For deterministic execution, a type of overhead comes from the calculation of the token (finding the task with minimal logical time). Current implementation requires O(N) time to decide which task should execute on next __det_start , where N is the number of threads. Every logical time update comes with such a computation process, more threads lead to more computation time. As we can see from all the benchmarks, deterministic system always leads to a higher overhead when thread count increases. In a previous discussion [23], it is pointed out that all the existing deterministic systems, regardless of what type of deterministic algorithm (lockstep or wait for turn), this kind of global communication is inevitable. This also implies that deterministic might not be practical for highly concurrent application with highly intensive inter-thread communications (synchronization primitives). Webservers like mongoose and nginx have very simple inter-thread communication , so that we achieved decent overhead during the benchmark. But for more complicated applications with more different types of locks, we might face severe performance overhead.

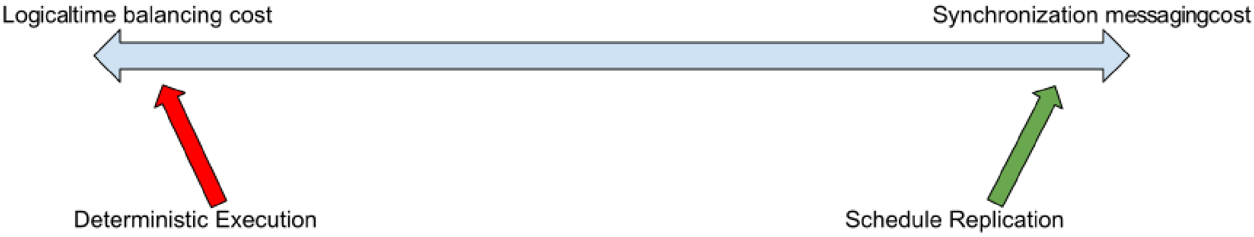

Figure 6.18: The tradeoff of two replication modes

6.5.3 Which Replication Mode Should I Use?

From all the benchmarks, we saw that some are better for Deterministic Execution and some are better for Schedule Replication. Figure 6.18 shows the basic idea of what kind of price we are paying for different replication modes. For Deterministic Execution, if we are able to precisely balance the logical time, or there is no logical time imbalance problem (like Nginx), it is the choice. Other than that, unless the cost of messaging becomes very significant, Schedule Replication is always the better choice due to its simplicity and efficiency.

Chapter 7

Conclusion

In this thesis we have explored a novel approach for doing replication: intra-machine replication with a multi-kernel OS. We have shown the challenges of replicating non-deterministic concurrent applications, and implemented two different replication modes to tame the non-deterministic thread-interleaving for concurrent applications. Also with a set of runtime support, we are able to replicate existing applications with minimal modification. With our benchmarks, we have also shown that both replication modes are able to achieve low overhead for different types of applications.

7.1 Contributions

This thesis presents the following contributions:

- **We implemented two different replication modes to synchronize the thread-interleavings and output for replicated concurrent applications.** Both of them achieved the same goal in two different directions: Deterministic Execution uses a deterministic algorithm to decide the order of execution on both primary and secondary; while Schedule Replication enforces the secondary replica to follow the non-deterministic execution order that happened previously on the primary kernel.

- **For Deterministic Execution, we developed a compiler framework to automatically instrument the application code to increase parallelism.** The compiler framework can profile the application and generate approximate values to increasing the logical time at the end of time consuming basic blocks. By balancing the logical time with instrumented values, the application can achieve decent overhead and scalability in Deterministic Execution.

- **Based on the common programming interface for both replication modes,**

we implemented a set of runtime support to eliminate the non-determinism in pthread library. By wrapping a code section with __det_start and __det_end system calls, the execution order of wrapped sections can be the same on both primary and secondary kernel. We implemented an instrumented pthread library that can be dynamically linked with LD_PRELOAD technique, which can minimize the effort of modifying the applications code. **For our evaluation, we did no modification to pbzip2, added 1 line to mongoose and added around 60 lines to nginx**.

- **We evaluated both replication modes with different concurrent applications. Our system showed decent overhead on both replication modes.** For a computational, non-network application we had **14.27% to 63.39%** slowdown for Deterministic Execution and maximum **0.89% to 36.3%** slowdown for Schedule Replication. For two web servers we had maximum **1.6% to 25.22%** slowdown for Deterministic Execution and maximum **0.23% to 1.96%** slowdown for Schedule Replication.

7.2 Future Work

7.2.1 Precise Tick Bump for Deterministic Execution

The major overhead in Deterministic Execution is the imbalanced logical time. The more precise the logical time incremental is the less waiting time we spend on __det_start . In chapter 3 we described our solution with a profiling approach, but from the evaluation we see this is still not precise enough to have the best performance. While performance counters are not deterministic enough to do the job [21], Intel PT [40] seems to be a promising approach to track the progress of the execution. PT is able to precisely track the actual behaviour of the execution (e.g. branch decisions) not just a counter of software events, as a result for the same executable, same input and same thread-interleaving, PT should always generate the same execution trace and be deterministic. With this dynamic tracing technique, we might be able to provide precise online tick bumping without having to instrument the code.

7.2.2 Per-Lock Synchronization

From the benchmark results, we occurred more overhead when the type of locks increases. This is because with serializing all the lock acquisitions, we are breaking the parallelism of accessing of different locks. If we only synchronize the access order of each particular lock while relaxing the total order of all the other locks, we might be able to achieve higher parallelism.

We could give an unique ID to __det_start as an additional parameter to identify different deterministic sections. However, the reason we couldn't implement this is that there is no perfect solution to generate this ID. A naive solution is to manually instrument the code with __det_start and __det_end and manually assign an ID to each deterministic section. This requires massive changes to existing code. Moreover, some applications require external libraries that includes lock primitives which makes this even harder.

On possible way is to use static analysis to identify the lock acquistions in the application. Midas [31] and CoreDet [41] both use compiler techniques to automatically instrument lock acquisitions. However with our experience on multiple modern applications' source code, there are applications tend to have their own wrapper of pthread lock functions, which makes it hard to identify different locks.

For pthread primitives, given the fact that most of them have futex involved, we could use the futex address [20] [42] as the unique IDs for each deterministic section. However during our test we found the primary and secondary replica cannot always have the same address for the same lock. This is expected because they are on separate Linux kernels and have different address spaces. Some deterministic systems address the problem of non-deterministic memory address allocation and mitigate the problem with special memory allocators [27] [17], in the future if we are able to synchronize the address space on the replicas, we might be able to directly use futex addresses as IDs for deterministic sections and achieve transparent per-lock synchronization.

7.2.3 Arbitrary Number of Replicas

Current implementation only supports one primary replica and one secondary replica. However Popcorn Linux supports booting arbitrary number of kernels as long as the hardware has enough resources. It would be interesting to explore the possibility of having more than one secondary replica. The major challenge is to decide the communication model for multiple replicas.

One solution is to do ”Chain-Messaging”. In this model we chain all the replicas one by one, the primary sends replication messages to the 2nd replica, and 2nd replica sends messages to 3rd replica and so on. Since point to point messaging is sure strictly FIFO on Popcorn Linux, this approach makes sure that all the replicas will see the same sequence of replication messages. Another advantage is that this can minimize the latency for the primary replica since it only communicates with one kernel. However in this approach the performance of the nth replica will be held back by all previous n-1th replicas, it cannot proceed until all its previous replicas have committed their operations.

We can also do broadcasting from the primary replica. Since we don't have the guarantee that message broadcasting in Popcorn Linux can behave in strictly FIFO, we might need to introduce consensus protocol such as Paxos [32] to make sure that all the replicas see the requests in the same order.

Another challenge is that when multiple replicas involved, who is going to be the primary when the previous primary fails. This can also be solved by utilizing the leader selection mechanism of Paxos [32].

7.2.4 Hybrid Replication

It is also interesting to extend this work to inter-machine and intra-machine hybrid replica. We can use intra-machine replica to deal with processor and memory faults with fast recovery, and inter-machine backup to deal with critical power failure with slower recovery. This will also have the same consensus problems as we mentioned in the previous section.

In our intra-machine implementation, we benefit a lot from our low latency messaging layer, the average time for sending a message is below 5 microseconds. However in a inter-machine setup, we might see different result than what we showed in our evaluation. The higher cost of inter-machine communication might cause greater slow down in Schedule Replication. While for Deterministic Execution which usually needs less messages, it might perform better than Schedule Replication in this case.

Bibliography

[1] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the Ninth European Conference on Computer Systems*, page 11. ACM, 2014.

[2] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 237–250, 2012.

[3] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 105–120. ACM, 2015.

[4] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, and Thomas C Bressoud. Practical and low-overhead masking of failures of tcp-based servers. *ACM Transactions on Computer Systems (TOCS)*, 27(2):4, 2009.

[5] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, Petros Maniatis, et al. Zeno: Eventually consistent byzantine-fault tolerance. In *NSDI*, volume 9, pages 169–184, 2009.

[6] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[7] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W Lee, Scott A Mahlke, and David I August. Runtime asynchronous fault tolerance via speculation. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 145–154. ACM, 2012.

[8] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M Chen, and Jason Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. *ACM SIGARCH Computer Architecture News*, 38(1):77–90, 2010.

[9] Thomas C Bressoud and Fred B Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[10] Jacob R Lorch, Andrew Baumann, Lisa Glendenning, Dutch Meyer, and Andrew Warfield. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 575–588, 2015.

[11] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[13] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel os based on linux. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.

[14] Benjamin H Shelton. *Popcorn Linux: enabling efficient inter-core communication in a Linux-based multikernel operating system.* PhD thesis, Virginia Polytechnic Institute and State University, 2013.

[15] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 85–96. ACM, 2009.

[16] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A Gibson, and Randal E Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 388–405. ACM, 2013.

[17] Tongping Liu, Charlie Curtsinger, and Emery D Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336. ACM, 2011.

[18] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108, 2009.

[19] Timothy Merrifield and Jakob Eriksson. Increasing concurrency in deterministic runtimes with conversion.

[20] Ulrich Drepper. Futexes are tricky. *Red Hat Inc, Japan*, 2005.

[21] Vincent M Weaver, Sally McKee, et al. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141–150. IEEE, 2008.

[22] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[23] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The deterministic execution hammer: How well does it actually pound nails. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming (WODET11)*, 2011.

[24] Charles E Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[25] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In *ACM Sigplan Notices*, volume 44, pages 81–96. ACM, 2009.

[26] Timothy Merrifield, Joseph Devietti, and Jakob Eriksson. High-performance determinism with total store order consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, page 31. ACM, 2015.

[27] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D Gribble. Deterministic process groups in dos. In *OSDI*, volume 10, pages 177–192, 2010.

[28] Cedomir Segulja and Tarek S Abdelrahman. Architectural support for synchronization-free deterministic parallel programming. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[29] Derek R Hower, Polina Dudnik, Mark D Hill, and David A Wood. Calvin: Deterministic or not? free will to choose. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 333–334. IEEE, 2011.

[30] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. Rcdc: a relaxed consistency deterministic computer. In *ACM SIGPLAN Notices*, volume 46, pages 67–78. ACM, 2011.

[31] Joseph G Slember and Priya Narasimhan. Static analysis meets distributed fault-tolerance: Enabling state-machine replication with nondeterminism. In *HotDep*, 2006.

[32] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[33] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[34] M Hill and M Xu Racey. A stress test for deterministic execution. http://pages.cs.wisc.edu/~markhill/racey.html.

[35] J Gilchrist. Parallel bzip2 (pbzip2) data compression software. *URL http://compression.ca/pbzip2*.

[36] Mongoose - embedded web server. https://github.com/cesanta/mongoose.

[37] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[38] Thread pools in nginx boost performance 9x! https://www.nginx.com/blog/thread-pools-boost-performance-9x/.

[39] Inside nginx: How we designed for performance & scale. https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/.

[40] Processor tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[41] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 53–64. ACM, 2010.

[42] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *AUUG Conference Proceedings*, page 85. AUUG, Inc., 2002.

MODULE *Det*

An algorithm for replicating multi-threaded applications as done in replicated Popcorn. The application is made deterministic through the use of logical time. Any inter-thread synchronization operation must be protected by calls to *EnterSync* and *ExitSync*. Reads of the socket *API* are modeled by *EnterRead*. The scheduler processes (one per kernel) makes sure that the different copies of the application are consistent.

$\text{EXTENDS } \mathit{Naturals}, \mathit{Sequences}, \mathit{Integers}, \mathit{Library}$

$\text{CONSTANTS } \mathit{Pid}, \mathit{MaxTime}, \mathit{Kernel}, \mathit{SchedulerPID}, \mathit{Requests}$

$\text{ASSUME } \mathit{SchedulerPID} \notin \mathit{Pid}$

$\mathit{InitLogTime} \triangleq 1$

$\mathit{LogTime} \triangleq \mathit{InitLogTime} \,..\, \mathit{MaxTime}$ The set of logical time values

Processes are of the form $\langle k, \mathit{pid} \rangle$, where k is the kernel the process is running on.

$P \triangleq \mathit{Kernel} \times \mathit{Pid}$

$\mathit{Primary} \triangleq \text{CHOOSE } k \in \mathit{Kernel} : \text{TRUE}$

$\mathit{Ker}(p) \triangleq p[1]$

$\mathit{PID}(p) \triangleq p[2]$

$\mathit{OnKernel}(\mathit{kernel}) \triangleq \{\mathit{kernel}\} \times \mathit{Pid}$

Logical time comparison, using *PIDs* to break ties.

$$\mathit{Less}(p, \mathit{tp}, q, \mathit{tq}) \triangleq$$
$$\quad \mathit{tp} < \mathit{tq} \lor (\mathit{tp} = \mathit{tq} \land \mathit{PID}(p) \leq \mathit{PID}(q))$$

The sequence of *TCP* packets that will be received. No duplicates allowed (therefore the set *TcpData* must be big enough) so that any misordering of the threads will lead to a different data read. For *TCPMultiStream*, each stream has different data.

$\mathit{StreamLength} \triangleq 3$

$\mathit{TcpData} \triangleq 1 \,..\, \mathit{StreamLength} * \mathit{Requests}$

$\mathit{TcpStream} \triangleq [i \in 1 \,..\, \mathit{StreamLength} \mapsto i]$

$$\mathit{TcpMultiStream} \triangleq [r \in 1 \,..\, \mathit{Requests} \mapsto$$
$$\qquad ([i \in 1 \,..\, \mathit{StreamLength} \mapsto i + (\mathit{StreamLength} * (r - 1))])]$$

$\text{ASSUME } \mathit{NoDup}(\mathit{TcpStream})$

$\text{ASSUME } \mathit{Len}(\mathit{TcpStream}) = \mathit{StreamLength}$

Shifts a sequence by 1: $\mathit{Shift}(\langle 1, 2, 3 \rangle) = \langle 2, 3 \rangle$ and $\mathit{Shift}(\langle\rangle) = \langle\rangle$.

```
Shift(s) ≜
    IF Len(s) > 1
     THEN [i ∈ 1 . . (Len(s) − 1) ↦ s[i + 1]]
     ELSE ⟨⟩
```

```
Shiftn(s) ≜
    IF Len(s) > 1
    THEN $[i \in 1 \,..\, (Len(s) - 1) \mapsto s[i + 1]]$
    ELSE $\langle -1 \rangle$
```

The algorithm *ReadAppend* models a set of worker threads being scheduled by the deterministic scheduler and executing the following code.

```
Code of worker $w$: $While(true)\{$
    $x = read(socket);$
    $append(\langle w,\, x \rangle,\, file);$
$\}$
```

Variables:

The variable bumps records all logical time bumps executed by the primary in order for the secondaries to do the same, *i.e.* the initial logical time, the new logical time, and the value read from the tcp buffer. $\langle t1,\, t2,\, d \rangle \in bumps[pid]$ means that the primary bumped process *pid* from logical time $t1$ to $t2$ and delivered the data d. Note that the scheduler set $bumps[pid]$ to a value that depends on the logical time of the processes on all replicas, and this value is then immediately available to all replicas. A more detailed model would instead include a distributed implementation of the choice of the logical time to bump the process to.

$reads[p]$ stores the last value read by p from the socket.

$tcpBuff[k]$ represents the state of the tcp buffer on kernel k. Each time a process reads from the buffer, the buffer shrinks by 1.

Definitions:

$Bumped(kernel)$ is the set of processes running on the kernel "kernel" which are waiting to execute a "bump" decided by the primary.

If $p \in Bumped(Ker(p))$ then $BumpedTo(p)$ is the logical time to which p should be bumped to.

If $p \in WaitingSync(Ker(p))$ then $IsNextProc(kernel,\, p)$ is true iff p is the process to be scheduled next, that is: (1) p has the lowest *ltime* among running and waiting-sync processes and (2) if q is on the same kernel and q is waiting for a read and the primary has already decided to which logical time tq to bump q, then $ltime[p]$ must be less than tq.

BumpTo is the logical time to which to bump a process that needs bumping. It is some logical time greater than all the logical times reached by any process on any kernel.

```
--algorithm $ReadAppend\{$
  variables
    $status = [p \in P \mapsto \text{"running"}]$,
    $ltime = [p \in P \mapsto InitLogTime]$,
    $file = [k \in Kernel \mapsto \langle\rangle]$,
    $bumps = [p \in Pid \mapsto \{\}]$,
    $reads\ \ = [p \in P \mapsto -1]$,
    $tcpBuff = [k \in Kernel \mapsto TcpMultiStream]$,
     Queue for accepted connections
    $socketQueue = [k \in Kernel \mapsto \langle\rangle]$,
     Queue for unhandled connections
    $requestQueue = [k \in Kernel \mapsto [r \in 1 \,..\, Requests \mapsto r]]$,
     The socket that is handled by a process
    $handledSocket = [p \in P \mapsto -1]$,
    $connections = [k \in Kernel \mapsto \langle\rangle]$

define {
    $Run(p) \triangleq status[p] = \text{"running"}$
    $Running(kernel) \triangleq \{p \in OnKernel(kernel) : Run(p)\}$
    $WaitingSync(kernel) \triangleq$
        $\{p \in OnKernel(kernel) : status[p] = \text{"waiting sync"}\}$
    $WaitingRead(kernel) \triangleq$
        $\{p \in OnKernel(kernel) : status[p] = \text{"waiting read"}\}$
    $Bumped(kernel) \triangleq \{p \in OnKernel(kernel) :$
        $\land\ status[p] = \text{"waiting read"}$
        $\land\ \exists\, t \in LogTime : \exists\, d \in TcpData : \langle ltime[p], t, d\rangle \in bumps[PID(p)]\}$
    $BumpedTo(p) \triangleq$
        $\text{CHOOSE } t \in LogTime : \exists\, d \in TcpData : \langle ltime[p], t, d\rangle \in bumps[PID(p)]$
    $BumpData(p) \triangleq$
        $\text{CHOOSE } d \in TcpData : \exists\, t \in LogTime : \langle ltime[p], t, d\rangle \in bumps[PID(p)]$
    $IsNextProc(kernel, p) \triangleq$
        $\land\ \forall\, q \in Running(kernel) \cup WaitingSync(kernel) :$
                $q \neq p \Rightarrow Less(p, ltime[p], q, ltime[q])$
        $\land\ \forall\, q \in Bumped(kernel) : Less(p, ltime[p], q, BumpedTo(q))$
    $BumpTo \triangleq \text{CHOOSE } i \in LogTime : \forall\, p \in P : ltime[p] < i$
}

macro $EnterRead(p)$ {
    $status[p] := \text{"waiting read"}$ ;
}
macro $EnterSync(p)$ {
    $status[p] := \text{"waiting sync"}$ ;
}
macro $ExitSync(p)$ {
    $ltime[p] := ltime[p] + 1$ ;
}

  Processes consume a connection
process $(worker \in P)$ {
          $ww1$: while (TRUE) {
                    $EnterSync(self)$ ;
          $ww2$:    await $Run(self)$ ;
          $ww3$:    if $(Len(requestQueue[Ker(self)]) > 0)$ {
                        $handledSocket[self] := requestQueue[Ker(self)][1]$ ;
                        $requestQueue[Ker(self)] := Shift(requestQueue[Ker(self)])$ ;
                    } ;
          $ww4$:    $ExitSync(self)$ ;
          $ww5$:    if $(handledSocket[self] \neq -1)$ {
                    $ww9$:    while $(Len(tcpBuff[Ker(self)][handledSocket[self]]) > 0)$ {
                                  $EnterRead(self)$ ;
                    $ww10$:  **await** $Run(self)$ ;
                             $EnterSync(self)$ ;
                    $ww11$:  **await** $Run(self)$ ;
                             $file[Ker(self)] :=$
                                 $Append(file[Ker(self)], \langle PID(self), reads[self]\rangle)$ ;
                    $ww12$:  $ExitSync(self)$ ;
                    }
                 } ;
          $ww13$:   $handledSocket[self] := -1$ ;
          } ;
     }

  **process** $(scheduler \in \{\langle k, SchedulerPID\rangle : k \in Kernel\})$ {
     $s1$:   **while** (TRUE){
               **either** {  Schedule a process waiting for synchronization:
                   **with** $(p \in \{p \in WaitingSync(Ker(self)) :$
                           $IsNextProc(Ker(self), p)\})$ {
                       $status[p] :=$ “running” }}
               **or** {  Bump a process waiting for a read:
                   **with** $(p \in WaitingRead(Ker(self)))$ {
                       **if** $(Ker(self) = Primary)$ {  On the primary.
                         Record the bump for the secondaries.
                           $bumps[PID(p)] :=$
                               $bumps[PID(p)] \cup \{\langle ltime[p], BumpTo, tcpBuff[Ker(self)][handledSocket[p]][1]$
                           $ltime[p] := BumpTo$ ;
                       }**else** {  On a replica:
 Wait until the primary has bumped $p$ and the data to be delivered to $p$ in at the head of the tcp buffer
                         **await** $p \in Bumped(Ker(self)) \wedge BumpData(p) = tcpBuff[Ker(self)][handledSocket$
                         $ltime[p] := BumpedTo(p)$ ;   Bump the process
                     } ;
                     $reads[p] := tcpBuff[Ker(self)][handledSocket[p]][1]$ ;
                     $tcpBuff[Ker(self)][handledSocket[p]] :=$
                         $Shift(tcpBuff[Ker(self)][handledSocket[p]])$ ;
                     $status[p] :=$ “running” ;   Let $p$ run.
                   }
               }
          }
     }
}
```

BEGIN TRANSLATION

VARIABLES $status$, $ltime$, $file$, $bumps$, $reads$, $tcpBuff$, $socketQueue$,
$requestQueue$, $handledSocket$, $connections$, pc

define statement

$Run(p) \triangleq status[p] =$ “running”

```
$Running(kernel) \triangleq \{p \in OnKernel(kernel) : Run(p)\}$
$WaitingSync(kernel) \triangleq$
    $\{p \in OnKernel(kernel) : status[p] = \text{“waiting sync”}\}$
$WaitingRead(kernel) \triangleq$
    $\{p \in OnKernel(kernel) : status[p] = \text{“waiting read”}\}$
$Bumped(kernel) \triangleq \{p \in OnKernel(kernel) :$
    $\land\ status[p] = \text{“waiting read”}$
    $\land\ \exists\, t \in LogTime : \exists\, d \in TcpData : \langle ltime[p],\ t,\ d\rangle \in bumps[PID(p)]\}$
$BumpedTo(p) \triangleq$
    $\textsc{choose}\ t \in LogTime : \exists\, d \in TcpData : \langle ltime[p],\ t,\ d\rangle \in bumps[PID(p)]$
$BumpData(p) \triangleq$
    $\textsc{choose}\ d \in TcpData : \exists\, t \in LogTime : \langle ltime[p],\ t,\ d\rangle \in bumps[PID(p)]$
$IsNextProc(kernel,\ p) \triangleq$
    $\land\ \forall\, q \in Running(kernel) \cup WaitingSync(kernel) :$
            $q \neq p \Rightarrow Less(p,\ ltime[p],\ q,\ ltime[q])$
    $\land\ \forall\, q \in Bumped(kernel) : Less(p,\ ltime[p],\ q,\ BumpedTo(q))$
$BumpTo \triangleq \textsc{choose}\ i \in LogTime : \forall\, p \in P : ltime[p] < i$


$vars \triangleq \langle status,\ ltime,\ file,\ bumps,\ reads,\ tcpBuff,\ socketQueue,$
           $requestQueue,\ handledSocket,\ connections,\ pc\rangle$

$ProcSet \triangleq (P) \cup (\{\langle k,\ SchedulerPID\rangle : k \in Kernel\})$

$Init \triangleq$  Global variables
          $\land\ status = [p \in P \mapsto \text{“running”}]$
          $\land\ ltime = [p \in P \mapsto InitLogTime]$
          $\land\ file = [k \in Kernel \mapsto \langle\rangle]$
          $\land\ bumps = [p \in Pid \mapsto \{\}]$
          $\land\ reads\ \ = [p \in P \mapsto\ -1]$
          $\land\ tcpBuff = [k \in Kernel \mapsto TcpMultiStream]$
          $\land\ socketQueue = [k \in Kernel \mapsto \langle\rangle]$
          $\land\ requestQueue = [k \in Kernel \mapsto [r \in 1\,..\,Requests \mapsto r]]$
          $\land\ handledSocket = [p \in P \mapsto\ -1]$
          $\land\ connections = [k \in Kernel \mapsto \langle\rangle]$
          $\land\ pc = [self \in ProcSet \mapsto \textsc{case}\ self \in P \rightarrow \text{“ww1”}$
                                          $\Box\ \ self \in \{\langle k,\ SchedulerPID\rangle : k \in Kernel\} \rightarrow \text{“s1”}]$

$ww1(self) \triangleq$  $\land\ pc[self] = \text{“ww1”}$
              $\land\ status' = [status\ \textsc{except}\ ![self] = \text{“waiting sync”}]$
              $\land\ pc' = [pc\ \textsc{except}\ ![self] = \text{“ww2”}]$
              $\land\ \textsc{unchanged}\ \langle ltime,\ file,\ bumps,\ reads,\ tcpBuff,\ socketQueue,$
                              $requestQueue,\ handledSocket,\ connections\rangle$

$ww2(self) \triangleq$  $\land\ pc[self] = \text{“ww2”}$
              $\land\ Run(self)$
              $\land\ pc' = [pc\ \textsc{except}\ ![self] = \text{“ww3”}]$
```

$$\begin{array}{l}
\land \textsc{unchanged}\ \langle \mathit{status},\ \mathit{ltime},\ \mathit{file},\ \mathit{bumps},\ \mathit{reads},\ \mathit{tcpBuff},\\
\qquad\qquad\quad \mathit{socketQueue},\ \mathit{requestQueue},\ \mathit{handledSocket},\\
\qquad\qquad\quad \mathit{connections}\rangle
\end{array}$$

$$\begin{array}{lll}
\mathit{ww}3(\mathit{self}) & \triangleq & \land\ \mathit{pc}[\mathit{self}] = \text{“ww3”}\\
& & \land\ \textsc{if}\ \mathit{Len}(\mathit{requestQueue}[\mathit{Ker}(\mathit{self})]) > 0\\
& & \qquad \textsc{then}\ \land\ \mathit{handledSocket}' = [\mathit{handledSocket}\ \textsc{except}\ ![\mathit{self}] = \mathit{requestQueue}[\mathit{Ker}(\mathit{self})][1]]\\
& & \qquad\qquad\quad \land\ \mathit{requestQueue}' = [\mathit{requestQueue}\ \textsc{except}\ ![\mathit{Ker}(\mathit{self})] = \mathit{Shift}(\mathit{requestQueue}[\mathit{Ker}(\\
& & \qquad \textsc{else}\ \ \ \land\ \textsc{true}\\
& & \qquad\qquad\quad \land\ \textsc{unchanged}\ \langle \mathit{requestQueue},\ \mathit{handledSocket}\rangle\\
& & \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“ww4”}]\\
& & \land\ \textsc{unchanged}\ \langle \mathit{status},\ \mathit{ltime},\ \mathit{file},\ \mathit{bumps},\ \mathit{reads},\ \mathit{tcpBuff},\\
& & \qquad\qquad\quad \mathit{socketQueue},\ \mathit{connections}\rangle
\end{array}$$

$$\begin{array}{lll}
\mathit{ww}4(\mathit{self}) & \triangleq & \land\ \mathit{pc}[\mathit{self}] = \text{“ww4”}\\
& & \land\ \mathit{ltime}' = [\mathit{ltime}\ \textsc{except}\ ![\mathit{self}] = \mathit{ltime}[\mathit{self}] + 1]\\
& & \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“ww5”}]\\
& & \land\ \textsc{unchanged}\ \langle \mathit{status},\ \mathit{file},\ \mathit{bumps},\ \mathit{reads},\ \mathit{tcpBuff},\ \mathit{socketQueue},\\
& & \qquad\qquad\quad \mathit{requestQueue},\ \mathit{handledSocket},\ \mathit{connections}\rangle
\end{array}$$

$$\begin{array}{lll}
\mathit{ww}5(\mathit{self}) & \triangleq & \land\ \mathit{pc}[\mathit{self}] = \text{“ww5”}\\
& & \land\ \textsc{if}\ \mathit{handledSocket}[\mathit{self}] \neq -1\\
& & \qquad \textsc{then}\ \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“ww9”}]\\
& & \qquad \textsc{else}\ \ \ \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“ww13”}]\\
& & \land\ \textsc{unchanged}\ \langle \mathit{status},\ \mathit{ltime},\ \mathit{file},\ \mathit{bumps},\ \mathit{reads},\ \mathit{tcpBuff},\\
& & \qquad\qquad\quad \mathit{socketQueue},\ \mathit{requestQueue},\ \mathit{handledSocket},\\
& & \qquad\qquad\quad \mathit{connections}\rangle
\end{array}$$

$$\begin{array}{lll}
\mathit{ww}9(\mathit{self}) & \triangleq & \land\ \mathit{pc}[\mathit{self}] = \text{“ww9”}\\
& & \land\ \textsc{if}\ \mathit{Len}(\mathit{tcpBuff}[\mathit{Ker}(\mathit{self})][\mathit{handledSocket}[\mathit{self}]]) > 0\\
& & \qquad \textsc{then}\ \land\ \mathit{status}' = [\mathit{status}\ \textsc{except}\ ![\mathit{self}] = \text{“waiting read”}]\\
& & \qquad\qquad\quad \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“ww10”}]\\
& & \qquad \textsc{else}\ \ \ \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“ww13”}]\\
& & \qquad\qquad\quad \land\ \textsc{unchanged}\ \mathit{status}\\
& & \land\ \textsc{unchanged}\ \langle \mathit{ltime},\ \mathit{file},\ \mathit{bumps},\ \mathit{reads},\ \mathit{tcpBuff},\ \mathit{socketQueue},\\
& & \qquad\qquad\quad \mathit{requestQueue},\ \mathit{handledSocket},\ \mathit{connections}\rangle
\end{array}$$

$$\begin{array}{lll}
\mathit{ww}10(\mathit{self}) & \triangleq & \land\ \mathit{pc}[\mathit{self}] = \text{“ww10”}\\
& & \land\ \mathit{Run}(\mathit{self})\\
& & \land\ \mathit{status}' = [\mathit{status}\ \textsc{except}\ ![\mathit{self}] = \text{“waiting sync”}]\\
& & \land\ \mathit{pc}' = [\mathit{pc}\ \textsc{except}\ ![\mathit{self}] = \text{“ww11”}]\\
& & \land\ \textsc{unchanged}\ \langle \mathit{ltime},\ \mathit{file},\ \mathit{bumps},\ \mathit{reads},\ \mathit{tcpBuff},\ \mathit{socketQueue},\\
& & \qquad\qquad\quad \mathit{requestQueue},\ \mathit{handledSocket},\ \mathit{connections}\rangle
\end{array}$$

$$\begin{array}{lll}
\mathit{ww}11(\mathit{self}) & \triangleq & \land\ \mathit{pc}[\mathit{self}] = \text{“ww11”}\\
& & \land\ \mathit{Run}(\mathit{self})\\
& & \land\ \mathit{file}' = [\mathit{file}\ \textsc{except}\ ![\mathit{Ker}(\mathit{self})] = \mathit{Append}(\mathit{file}[\mathit{Ker}(\mathit{self})],\ \langle \mathit{PID}(\mathit{self}),\ \mathit{reads}[\mathit{self}]\rangle)]
\end{array}$$

```
                     ∧ pc′ = [pc EXCEPT ![self] = "ww12"]
                     ∧ UNCHANGED ⟨status, ltime, bumps, reads, tcpBuff,
                                  socketQueue, requestQueue, handledSocket,
                                  connections⟩

ww12(self) ≜  ∧ pc[self] = "ww12"
              ∧ ltime′ = [ltime EXCEPT ![self] = ltime[self] + 1]
              ∧ pc′ = [pc EXCEPT ![self] = "ww9"]
              ∧ UNCHANGED ⟨status, file, bumps, reads, tcpBuff, socketQueue,
                           requestQueue, handledSocket, connections⟩

ww13(self) ≜  ∧ pc[self] = "ww13"
              ∧ handledSocket′ = [handledSocket EXCEPT ![self] = −1]
              ∧ pc′ = [pc EXCEPT ![self] = "ww1"]
              ∧ UNCHANGED ⟨status, ltime, file, bumps, reads, tcpBuff,
                           socketQueue, requestQueue, connections⟩

worker(self) ≜ ww1(self) ∨ ww2(self) ∨ ww3(self) ∨ ww4(self)
                  ∨ ww5(self) ∨ ww9(self) ∨ ww10(self) ∨ ww11(self)
                  ∨ ww12(self) ∨ ww13(self)

s1(self) ≜  ∧ pc[self] = "s1"
            ∧ ∨ ∧ ∃ p ∈    {p ∈ WaitingSync(Ker(self)) :
                            IsNextProc(Ker(self), p)} :
                      status′ = [status EXCEPT ![p] = "running"]
                  ∧ UNCHANGED ⟨ltime, bumps, reads, tcpBuff⟩
              ∨ ∧ ∃ p ∈ WaitingRead(Ker(self)) :
                      ∧ IF Ker(self) = Primary
                           THEN  ∧ bumps′ = [bumps EXCEPT ![PID(p)] = bumps[PID(p)] ∪ {⟨ltime[p], Bum
                                 ∧ ltime′  = [ltime EXCEPT ![p] = BumpTo]
                           ELSE  ∧ p ∈ Bumped(Ker(self)) ∧ BumpData(p) = tcpBuff[Ker(self)][handledSoc
                                 ∧ ltime′  = [ltime EXCEPT ![p] = BumpedTo(p)]
                                 ∧ bumps′ = bumps
                      ∧ reads′ = [reads EXCEPT ![p] = tcpBuff[Ker(self)][handledSocket[p]][1]]
                      ∧ tcpBuff′ = [tcpBuff EXCEPT ![Ker(self)][handledSocket[p]] = Shift(tcpBuff[Ker(self)
                      ∧ status′ = [status EXCEPT ![p] = "running"]
            ∧ pc′ = [pc EXCEPT ![self] = "s1"]
            ∧ UNCHANGED ⟨file, socketQueue, requestQueue, handledSocket,
                         connections⟩

scheduler(self) ≜ s1(self)

Next ≜ (∃ self ∈ P : worker(self))
          ∨ (∃ self ∈ {⟨k, SchedulerPID⟩ : k ∈ Kernel} : scheduler(self))

Spec ≜ Init ∧ □[Next]_vars
```

END TRANSLATION

$$
\begin{array}{l}
\mathit{RequestSynchronized} \triangleq \forall\, k1,\ k2 \in \mathit{Kernel} : \\
\quad \lor\ \mathit{Prefix}(\mathit{socketQueue}[k1],\ \mathit{socketQueue}[k2]) \\
\quad \lor\ \mathit{Prefix}(\mathit{socketQueue}[k2],\ \mathit{socketQueue}[k1])
\end{array}
$$

$$
\begin{array}{l}
\mathit{FilesSynchronized} \triangleq \forall\, k1,\ k2 \in \mathit{Kernel} : \\
\quad \lor\ \mathit{Prefix}(\mathit{file}[k1],\ \mathit{file}[k2]) \\
\quad \lor\ \mathit{Prefix}(\mathit{file}[k2],\ \mathit{file}[k1])
\end{array}
$$

$$
\begin{array}{l}
\mathit{ConnectionSynchronized} \triangleq \forall\, k1,\ k2 \in \mathit{Kernel} : \\
\quad \lor\ \mathit{Prefix}(\mathit{connections}[k1],\ \mathit{connections}[k2]) \\
\quad \lor\ \mathit{Prefix}(\mathit{connections}[k2],\ \mathit{connections}[k1])
\end{array}
$$