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: Replication, Runtime Systems, Concurrent Programming, Multi-kernel Operating System

Replication of Concurrent Applications in a Shared Memory Multikernel

Yuzhong Wen

(ABSTRACT)

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc nec elit molestie, mattis mi a, consequat arcu. Fusce venenatis rhoncus elit. Morbi ornare, libero a bibendum pretium, nibh orci tristique mauris, in suscipit mauris nibh ac metus. Nullam in sem vitae nisi aliquet iaculis in a nibh. Aliquam lobortis quis turpis ut tempus. Sed eu sapien eu nisi placerat viverra pharetra eu turpis. Mauris placerat massa mi, auctor facilisis sem consequat in. Pellentesque sollicitudin placerat mi quis rhoncus. In euismod lorem semper, scelerisque leo et, dapibus diam. Suspendisse augue dui, placerat at finibus a, cursus vitae erat. Nam accumsan magna vitae lorem tincidunt, et rhoncus elit consequat.

Suspendisse ut tellus at ex suscipit sollicitudin ut ut elit. Nam malesuada molestie elit eget luctus. Donec id quam ullamcorper, aliquam mauris at, congue felis. Nunc dapibus dui sit amet nisl laoreet, eget rhoncus est tempor. Mauris in blandit mauris. Aenean vitae ipsum lacinia, blandit turpis et, feugiat purus. Mauris in finibus quam, ac dictum lorem. Nam dignissim luctus ante. Suspendisse risus felis, imperdiet a lobortis sed, suscipit ac dui. Nullam fermentum velit eu congue dictum. Pellentesque tempor dui vel nisl tristique, non sollicitudin odio elementum. In ultricies elementum mattis.

Vestibulum eget imperdiet eros. Proin bibendum sit amet felis quis dignissim. Aliquam convallis mauris ut sapien gravida, eu consequat lacus dignissim. Vivamus porttitor hendrerit nisl, sit amet suscipit lorem vestibulum ut. Donec id tellus condimentum, sollicitudin sapien vel, lobortis nulla. Donec et elit quis est tempor semper. Aliquam erat volutpat. In nec consectetur dui. Nullam aliquam diam at eros ultrices vehicula. Nulla nibh ex, condimentum vitae nisl sed, aliquet ultricies sapien. Suspendisse potenti. Suspendisse pellentesque tincidunt facilisis. Morbi sodales vulputate ex malesuada molestie. Vestibulum eget placerat nunc.

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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

State machine replication (SMR) has been widely used for fault-tolerance purpose in nowadays computing services. In SMR, it models the service 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 result. 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 state machines will get diverged in different states even with the same input set.

Current State Machine Replication approaches

Chapter 2

Popcorn Linux Background

Our replication prototype is built on top of Popcorn Linux. It is a multi-kernel OS which allows a multi-core system to boot multiple Linux kernels.

2.1 Hardware Partitioning

In Popcorn Linux, hardware resources are partitioned into arbitrary divisions, each booted kernel instance can have the full control of its own partition.

- CPU Partitioning: Popcorn Linux is able to map an arbitrary number of CPU cores to each kernel instance. In order to get the maximum performance for concurrent applications we prefer to evenly assign CPU cores to each kernel.
- Memory Partitioning: By setting the starting address and memory range during the boot time of a kernel, Popcorn Linux can also partition the memory resources for all the booted kernel.

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. When a critical hardware error happens on one kernel's hardware partition, this isolation guarantees that the error won't get propagated to another.

2.2 Inter-Kernel Messaging Layer

2.3 Popcorn Namespace

2.3.1 Replicated Execution

2.3.2 FT PID

2.4 Network Stack Replication

Chapter 3

Shogoki: Deterministic Execution System

Deterministic execution 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[1]. 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[2][3][4]. 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, but unlike other runtime deterministic systems, our runtime does not directly trap pthread primitives, but provides two system calls for programmer to define a deterministic section. The runtime 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.

This chapter is structured as follows:

- Section 3.1 shows the basic algorithm and programming interface of the deterministic system.
- Section 3.2 explains the logical time imbalance problem of this algorithm and two

solutions for two different cases.

3.1 Logical Time Based Deterministic Scheduling

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

```
1  void producer() {
2      while (running){
3          item = generate_item();
4          syscall(__NR_det_start);
5          pthread_mutex_lock(mutex);
6          syscall(__NR_det_end);
7          putItem(queue, item);
8          pthread_mutex_unlock(mutex);
9      }
10 }
11
12 void consumer() {
13     while (running){
14         syscall(__NR_det_start);
15         pthread_mutex_lock(mutex);
16         syscall(__NR_det_end);
17         item = getItem(queue);
18         pthread_mutex_unlock(mutex);
19         consume_item(item);
20     }
21 }
```

Figure 3.1: An example use of the deterministic syscalls

```
1  void __det_start()
2  {
3      if (token->token != current)
4          sleep(current);
5      current->ft_det_state = FT_DET_ACTIVE;
6  }
7  void __det_end()
8  {
9      current->ft_det_state = FT_DET_INACTIVE;
10     __update_tick(1);
11 }
12 void __det_tick(int tick)
13 {
14     __update_tick(tick);
15 }
16 void __update_tick(int tick)
17 {
18     current->tick += tick;
19     token->task = find_task_with_min_tick(ns);
20     if (is_waiting_for_toUponken(token->task))
21         wake_up(token->task);
22 }
```

Figure 3.2: Simplified implementation of deterministic system calls

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 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 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.

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

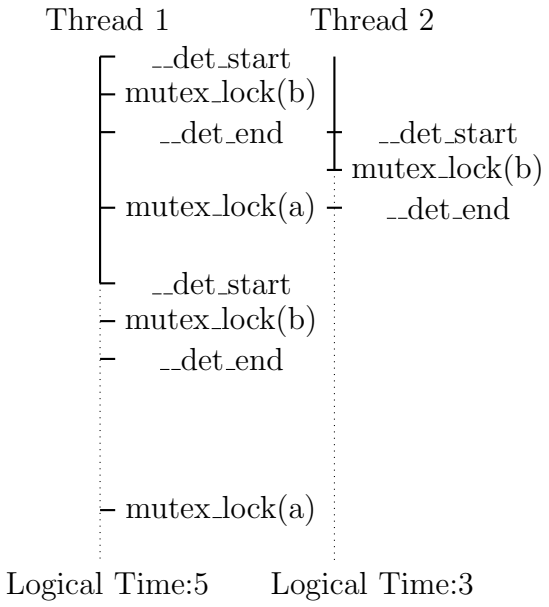

Figure 3.3: An example of deadlock

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.

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, 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:

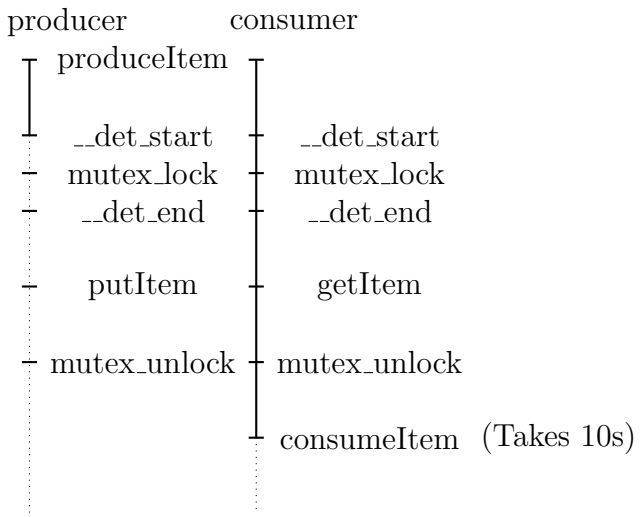

Figure 3.4: An example of logical time imbalance.

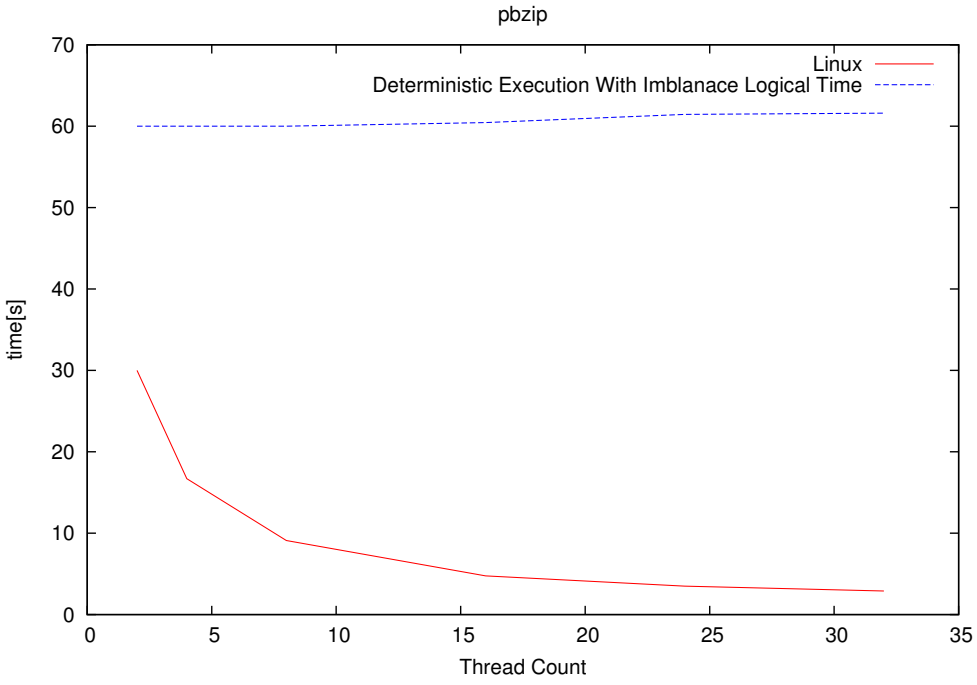

Figure 3.5: pbzip2 without logical time balancing

- __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, 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.

```
1  if.end.23:                                        ; preds = %for.end
2    %38 = load i8*, i8** %CompressedData, align 8
3    %39 = call i32 (i32, ...) @bbprof_start(i32 249)
4    %40 = load %struct.outBuff*, %struct.outBuff** %fileData, align 8
5    %buf = getelementptr inbounds %struct.outBuff, %struct.outBuff* %40,
         i32 0, i32 0
6    %41 = load i8*, i8** %buf, align 8
7    %42 = load %struct.outBuff*, %struct.outBuff** %fileData, align 8
8    %bufSize24 = getelementptr inbounds %struct.outBuff, %struct.outBuff*
          %42, i32 0, i32 1
9    %43 = load i32, i32* %bufSize24, align 4
10   %44 = load i32, i32* @_ZL12BWTblockSize, align 4
11   %45 = load i32, i32* @_ZL9Verbosity, align 4
12   %call25 = call i32 @BZ2_bzBuffToBuffCompress(i8* %38, i32* %outSize,
         i8* %41, i32 %43, i32 %44, i32 %45, i32 30)
13   store i32 %call25, i32* %ret, align 4
14   %46 = load i32, i32* %ret, align 4
15   %cmp26 = icmp ne i32 %46, 0
16   %47 = call i32 (i32, ...) @bbprof_end(i32 249)
17   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.

```
1    (.....)
2    %bufSize24 = getelementptr inbounds %struct.outBuff, %struct.outBuff*
          %35, i32 0, i32 1
3    %36 = load i32, i32* %bufSize24, align 4
4    %37 = load i32, i32* @_ZL12BWTblockSize, align 4
5    %38 = load i32, i32* @_ZL9Verbosity, align 4
6    %call25 = call i32 @BZ2_bzBuffToBuffCompress(i8* %32, i32* %outSize,
         i8* %34, i32 %36, i32 %37, i32 %38, i32 30)
7    store i32 %call25, i32* %ret, align 4
8    %39 = load i32, i32* %ret, align 4
9    %cmp26 = icmp ne i32 %39, 0
10   %40 = call i32 (...) @syscall(i32 321, i64 2895535)
11   br i1 %cmp26, label %if.then.27, label %if.end.29
```

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

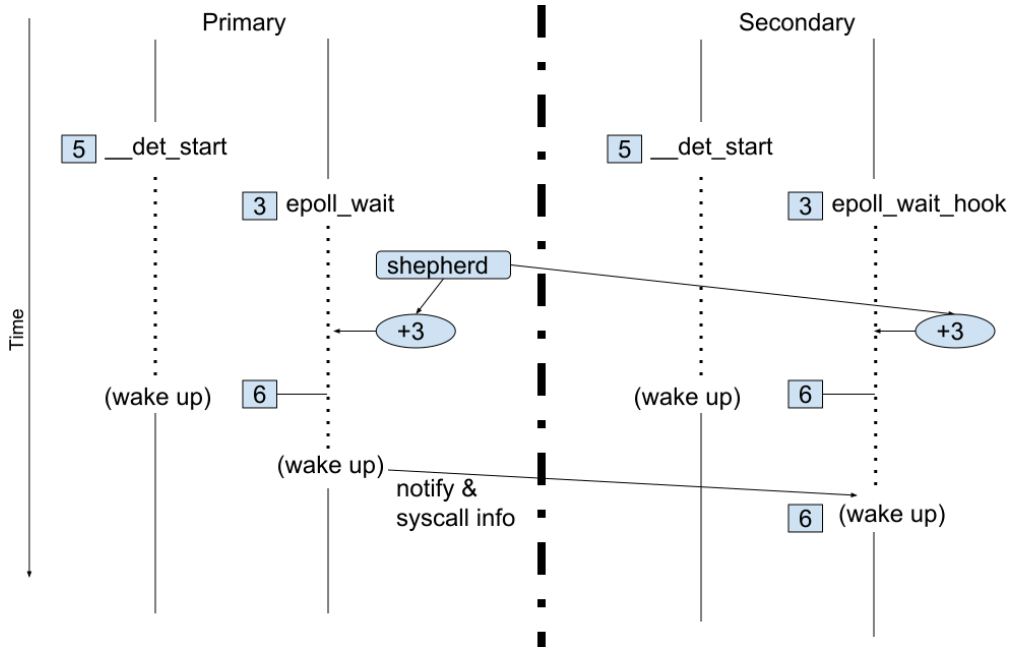

Figure 3.8: An example of tick bumping

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 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.

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 [**?**], 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++ [5] 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 [6] 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[4], Parrot[2] and Dthreads[3] 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-

```
1  while (!kthread_should_stop()) {
2    if (ns->task_count == 0 ||
3      ns->wait_count == 0) {
4      sleep(); // Sleep until some task wakes it up
5      continue;
6    }
7    token = ns->token;
8    tick = token->task->ft_det_tick;
9    udelay(20); // delay for a small duration
10   token2 = ns->token;
11   tick2 = token2->task->ft_det_tick;
12   // Which means the token hasn't been changed during the delay,
13   // It's time to bump the tick
14   if (token == token2 && tick2 == tick) {
15     if (!is_waiting_for_token(token->task) &&
16       (is_concerned_syscall(token->task->current_syscall)) {
17         if (ns->wait_count != 0 &&
18           token->task->bumped == 0) {
19             bump_task = token->task;
20             id_syscall = token->task->id_syscall;
21             bump = ns->last_tick + 1;
22             previous_bump = token->task->ft_det_tick;
23             token->task->ft_det_tick = ns->last_tick + 1;
24             update_token(ns);
25             send_bump(bump_task, id_syscall, previous_bump, bump);
26             continue;
27          }
28       }
29    }
30 }
```

Figure 3.9: Simplified implementation of Tick Shepherd

deterministic[7], 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. dOS [8] and provides an OS layer to make any program running on top of it deterministic, which is applicable for all kinds of parallel programming models. However dOS's overhead is too high due to massive trapping to shared memory accesses, it is not practical for high performance applications. DMP [1] 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 [9] and [**?**], they both proposed architectural solution 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 [10] 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 don't 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 didn't mention network operations(Dthreads[3]) or simply skip the threads doing such operations (Kendo[4], Parrot[2]), 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 [8] 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 [11] utilizes Parrot[2] as the underlying deterministic system but without skipping the network operations. On top of that, Crane also 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.

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 are going to describe Schedule Replication for replicated applications. In this algorithm, 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's revisit some important properties that are provided by the deterministic system.

- Serialization of deterministic areas. (The code region between detstart and detend).
- Same total order of getting into deterministic areas on primary and secondary.

```
1  /*
2   * Definitions:
3   * ns: current popcorn namespace
4   * ns->global_mutex: global_mutex in current namespace
5   * ns->seq: global sequence number Seq_global
6   * current->seq: task sequence number Seq_thread
7   * current->ft_pid: replicated task unique identifier
8   */
9  void __det_start()
10 {
11     if (is_secondary(current))
12         wait_for_sync(current->seq,
13             ns->seq, current->ft_pid);
14     lock(ns->global_mutex);
15     current->ft_det_state = FT_DET_ACTIVE;
16 }
17 void __det_end()
18 {
19     if (is_primary(current))
20         send_sync(current->seq,
21             ns->seq, current->ft_pid);
22     current->seq++;
23     ns->seq++;
24     current->ft_det_state = FT_DET_INACTIVE;
25     unlock(ns->global_mutex);
26 }
```

Figure 4.1: Simplified implementation of system calls for schedule replication

The first property is guaranteed by the fact that the logical time won't 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}.

- 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.

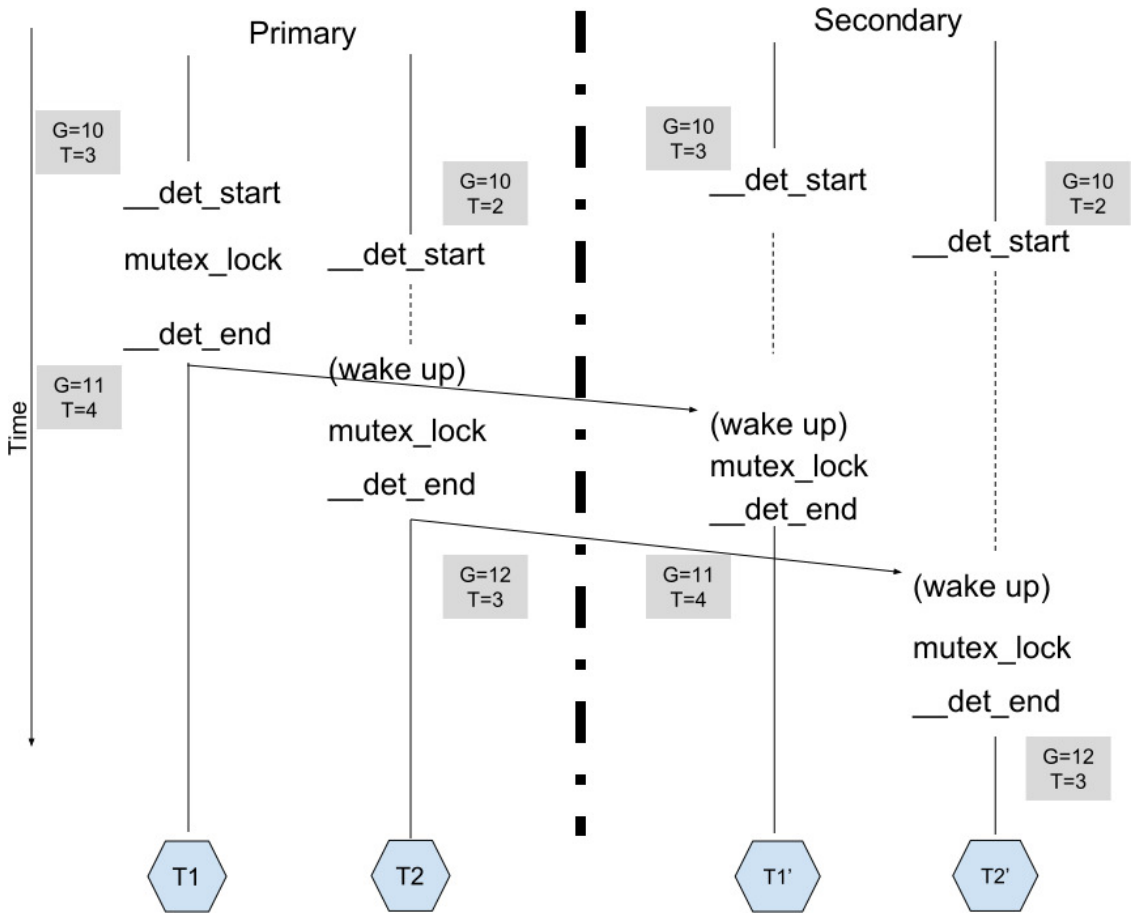

Figure 4.2: An example of Schedule Replication

4.2 Related Work

Several previous works also presented the idea that not using a deterministic system for replication.

4.2.1 Replication

4.2.2 Execute And Verify

4.2.3 Record And Replay

Partial order

Execute and verify

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.

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 doesn't 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 didn't 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 it's 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 secondary. As shown in Figure 5.1, the output of poll is the fds array and the return value.

```
1 int poll(struct pollfd *fds, nfds_t nfds, int timeout);
2
3 struct pollfd {
4     int   fd;         /* file descriptor */
5     short events;     /* requested events */
6     short revents;    /* returned events */
7 };
```

Figure 5.1: poll prototype and pollfd data structure

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 that uses pthread, it will be even more troublesome to recompile the needed external library.

```
1  int epoll_wait(int epfd, struct epoll_event *events,
2                       int maxevents, int timeout);
3
4  typedef union epoll_data {
5      void    *ptr;
6      int      fd;
7      uint32_t u32;
8      uint64_t u64;
9  } epoll_data_t;
10
11 struct epoll_event {
12     uint32_t     events;    /* Epoll events */
13     epoll_data_t data;      /* User data variable */
14 };
```

Figure 5.2: epoll_wait prototype and epoll_event data structure

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.

```
1  int pthread_mutex_lock(pthread_mutex_t *mutex)
2  {
3      int ret;
4      static int (*pthread_mutex_lock_real)(pthread_mutex_t *mutex) =
           NULL;
5      if (!handle) {
6          handle = dlopen(PTHREAD_PATH, RTLD_LAZY);
7      }
8      if (!pthread_mutex_lock_real)
9          pthread_mutex_lock_real = dlsym(handle, "pthread_mutex_lock");
10
11     syscall(__NR_det_start);
12     ret = pthread_mutex_lock_real(mutex);
13     syscall(__NR_det_end);
14
15     return ret;
16 }
```

Figure 5.3: pthread_mutex_lock in the LD_PRELOAD library

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.

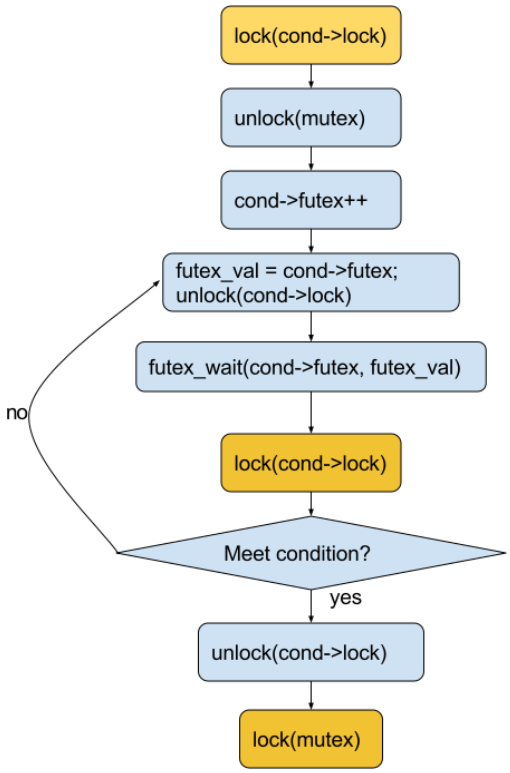

Figure 5.4: glibc pthread_cond_wait internal work flow

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.

For pthread_cond_timed_wait, we did the same re-implementation for it. However the timeout part is a little bit tricky.

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?
- Breakdown: Where does the overhead come from?

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 a 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 [12] 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 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 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

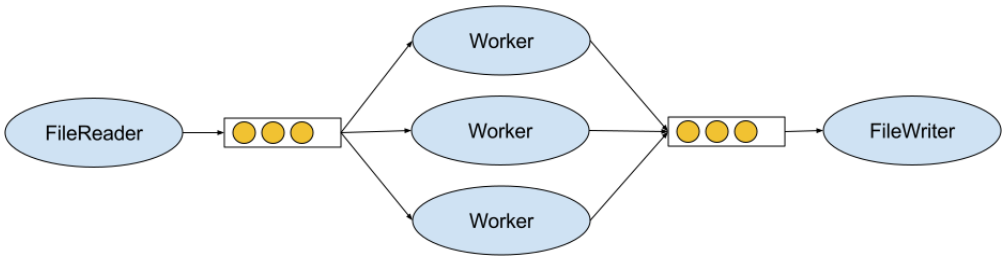

Figure 6.1: pbzip2 concurrent model

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.

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 become more obvious.

The result for deterministic execution seems more interesting. For all the benchmark runs, there seems to be a constant 2 3 seconds overhead compare to vanilla Linux. This is due to

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

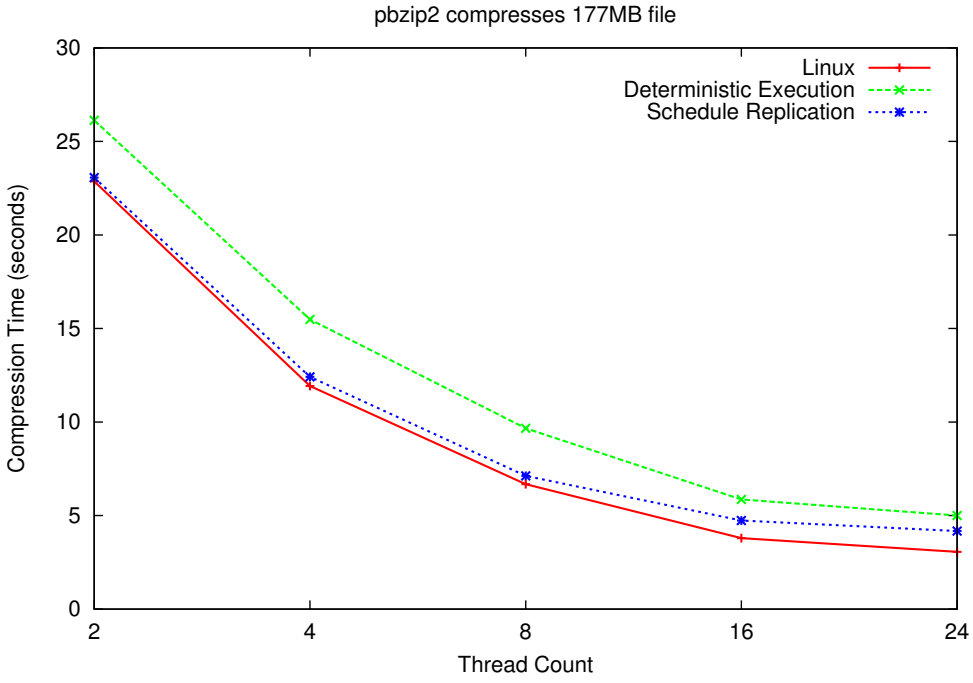

Figure 6.2: pbzip2 performance

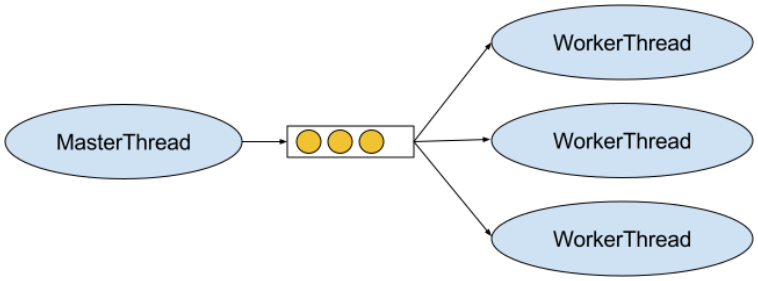

Figure 6.3: mongoose concurrent model

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

6.3 Mongoose Webserver

Mongoose is a compact multithreaded webserver. The concurrent model is shown in Figure 6.3. 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 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. Given mongoose's concurrent model, we can see that different file size will affect the frequency of acquiring locks. In this way we can figure out how will two algorithm perform under different lock acquisition load. Also, different thread counts can reflect the scalability of our system.

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

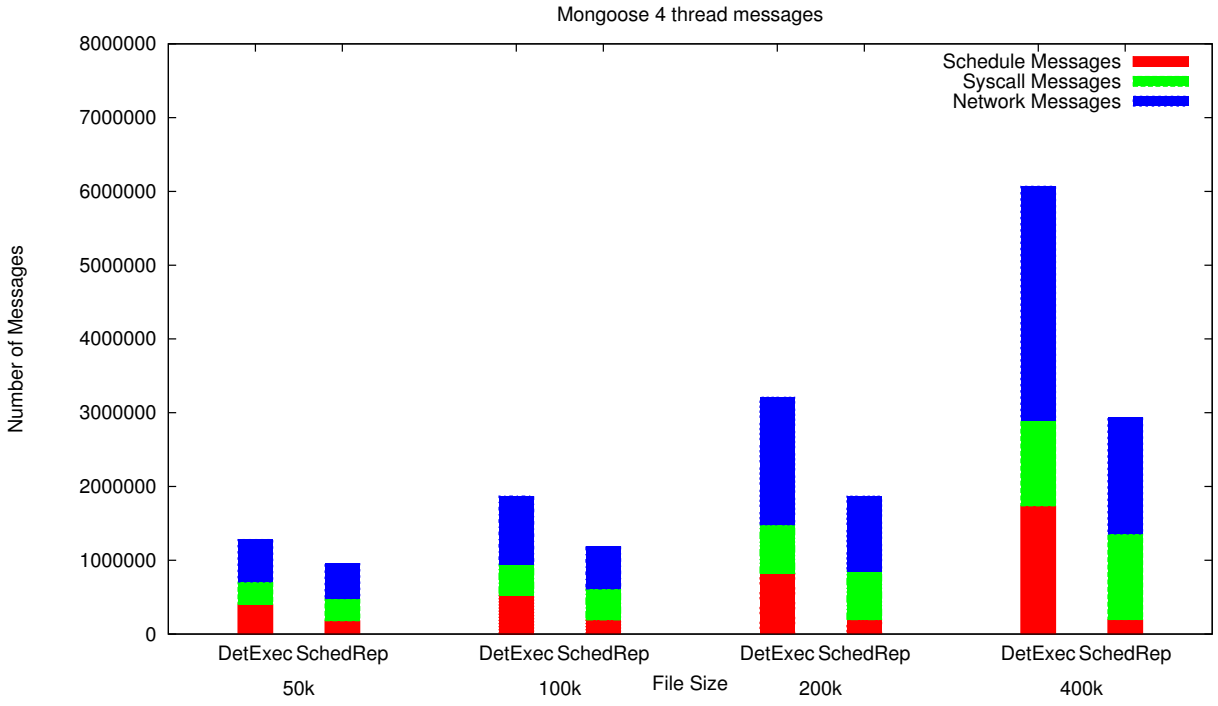

Figure 6.4: mongoose messages in 4 threads

6.3.1 Results

6.3.2 Overhead Breakdown

6.3.3 Message Breakdown

In order to measure the message consumption of the benchmark, we put counters in different subsystems to count how many messages we need for each benchmark. Figure 6.4, Figure 6.5 and Figure 6.6 show the breakdown of overall messages for each benchmark set. 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.

An interesting result is that actually we need much more messages for deterministic execution, which contradicts the assumption we made for Deterministic Execution. 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.

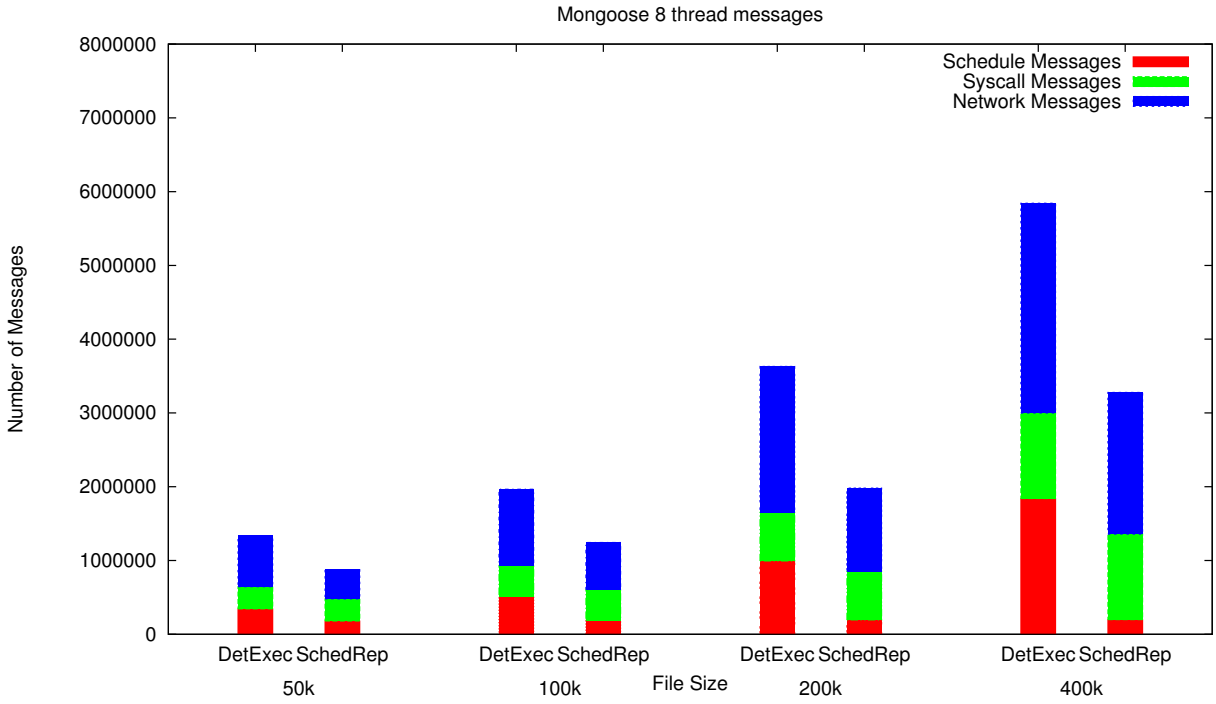

Figure 6.5: mongoose messages in 8 threads

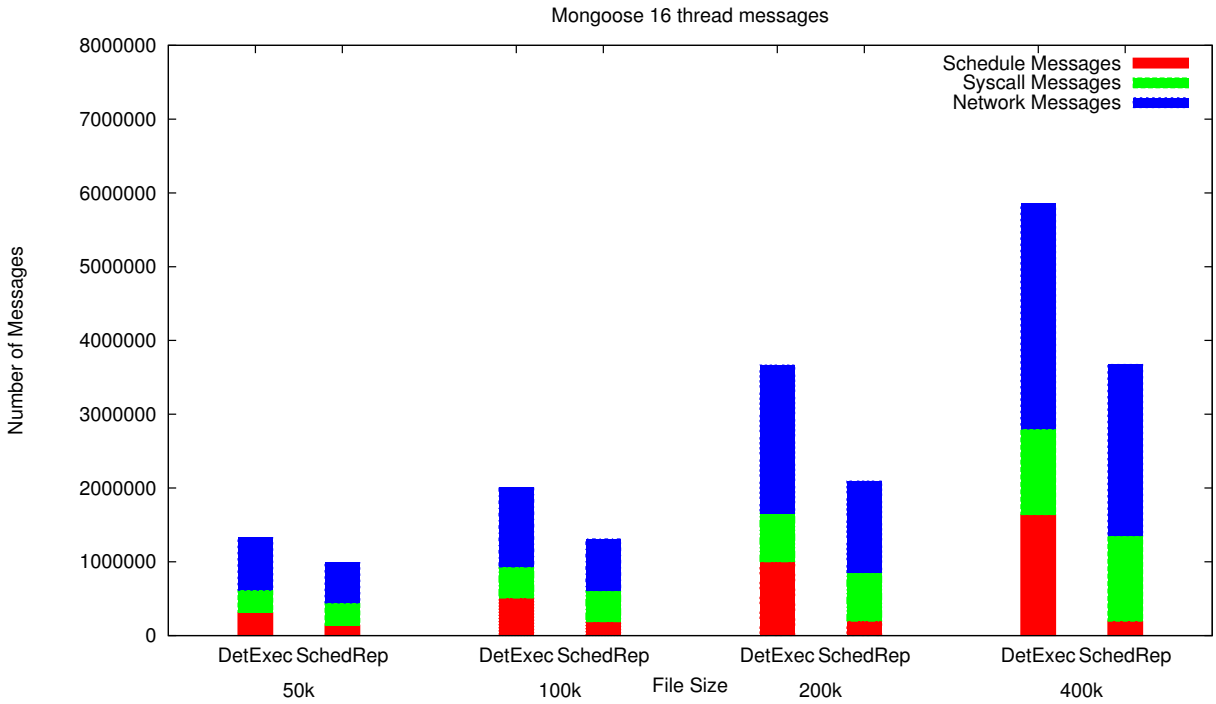

Figure 6.6: mongoose messages in 16 threads

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

Table 6.4: Tracked system calls used by nginx

Client count	Deterministic Execution	Schedule Replication
2	30.38%	11.93%
16	41.05%	26.94%
64	39.66%	26.48%

Table 6.5: Redis Overall Overhead of Each Replication Mode

6.4 Nginx Webserver

Nginx is a sophisticated webserver with multiple threading modes.

6.5 Redis Database Server

Redis is an in-memory database server. It uses a single thread to process requests, but it dynamically creates new threads to write the in-memory data to the disk. This benchmark is perfect for stressing the flexibility of dealing with dynamically spawned threads.

For the performance test we used the redis-benchmark tool, we used the default benchmark parameter which will test all the operations. Each operation is tested for 10000 requests. We also have different number of concurrent clients to stress the server with different frequency of requests. We ran each setup for 5 times and took the average of the numbers.

Redis uses an alternative memory allocator jemalloc, which contains some internal locks to ensure mutual exclusion for concurrent memory allocation. As mentioned in Section **??**, those lock acquisitions doesn't affect the output at all, so we modified the jemalloc's source code, to skip the synchronizations for those locks.

6.5.1 Results

Figure 6.7, figure 6.8 and figure 6.9 show the performance of redis with 2, 16 and 64 concurrent clients. Table 6.5 shows the overall overhead of each replication mode.

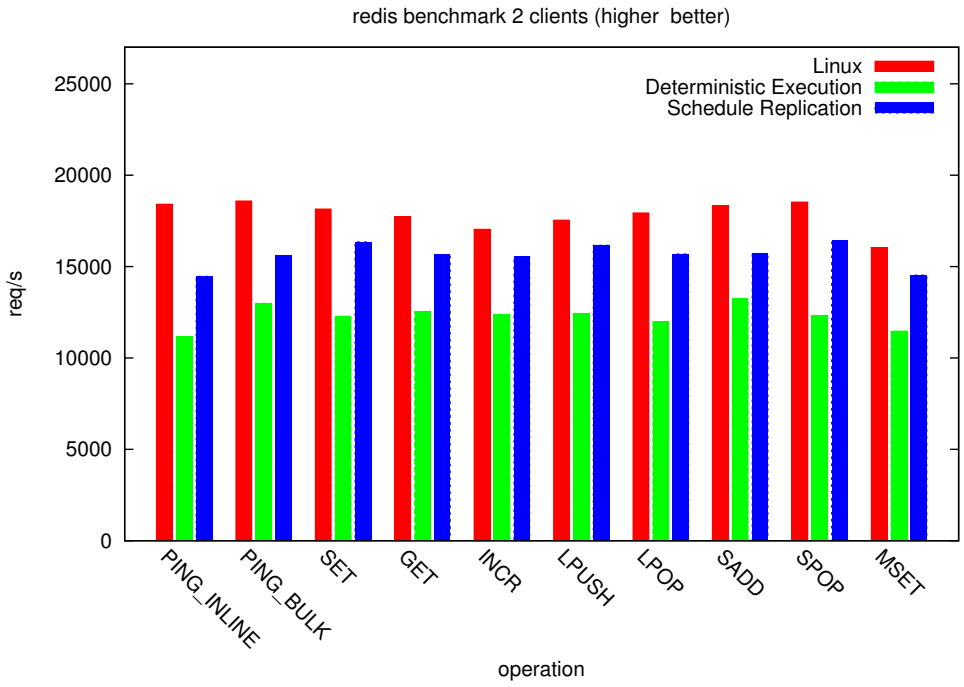

Figure 6.7: redis benchmark with 10000 requests and 2 clients

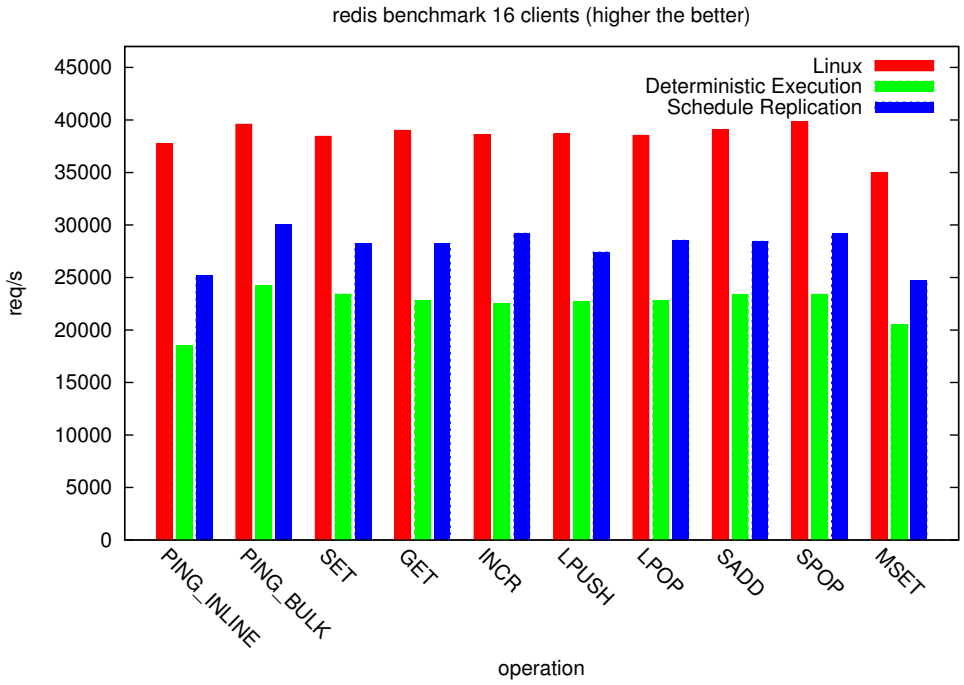

Figure 6.8: redis benchmark with 10000 requests and 16 clients

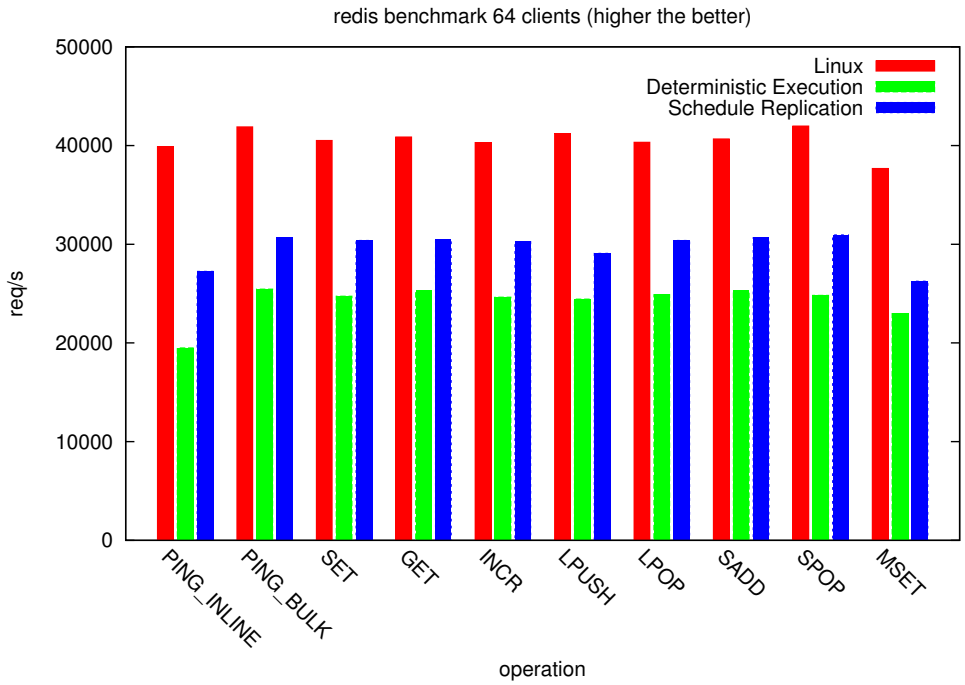

Figure 6.9: redis benchmark with 10000 requests and 64 clients

6.5.2 Overhead Profiling

6.6 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 of concurrent models and different application types. Here we will discuss some particular performance issues, and also some problems during the benchmark.

6.6.1 Benchmark for Nginx's Multi-process Mode

So far we only showed benchmarks for multi-threaded applications. However our system also supports multi-process applications. 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, which is lock across processes, to ensure an incoming request only wakes up one waiting worker. In order to figure 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 in the same state all the time. However no matter how many workers we

put, it was always only one worker hands all the request. This is due to a relatively low workload we used in our benchmark. However, because of the limitation of both our network card (1Gbps) and the limitation of the network stack replication, we couldn't saturate the worker.

Although this benchmark showed the correctness of our system, the unbalanced load seems not making any sense in terms of scalability. As a result we didn't include the result in this chapter. In the future with the optimization of the network stack replication and a better network card, we might be able show a more reasonable result for this benchmark.

6.6.2 Deterministic Execution's Dilemma

For deterministic execution, a type of overhead comes from the calculation of the token. 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 leads to more calculation 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 [**?**], 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.

6.6.3 Benefit From Messaging Layer

Chapter 7

Conclusion

7.1 Contributions

7.2 Future Work

7.2.1 Pre-Lock Synchronization

7.2.2 Arbitrary Number Replicas

7.2.3 Hybrid Replication

7.3 Further Evaluation

Bibliography

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] M Hill and M Xu Racey. A stress test for deterministic execution.