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: blah, blah

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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

William Shakespeare has profoundly affected the field of literature worldwide. In the United States there was a surge of Shakespearean literature starting in the 1960s, with the opening of the Montgomery Shakespearean festival and continuing into the present ...

Table 1.1: The Graduate School wants captions above the tables.

x	1	2
1	1	2
2	2	4

Chapter 2

Popcorn Linux Background

2.1   Multikernel Boot

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[?]. 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[?][?][?]. 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

Thread 1   Thread 2

```
┌    __det_start
├  mutex_lock(b)  │
├    __det_end    ┤    __det_start
                  ├  mutex_lock(b)
├  mutex_lock(a)  ┤    __det_end
│
└    __det_start
├  mutex_lock(b)
├    __det_end
│
│
├  mutex_lock(a)
```

Logical Time:5   Logical Time:3
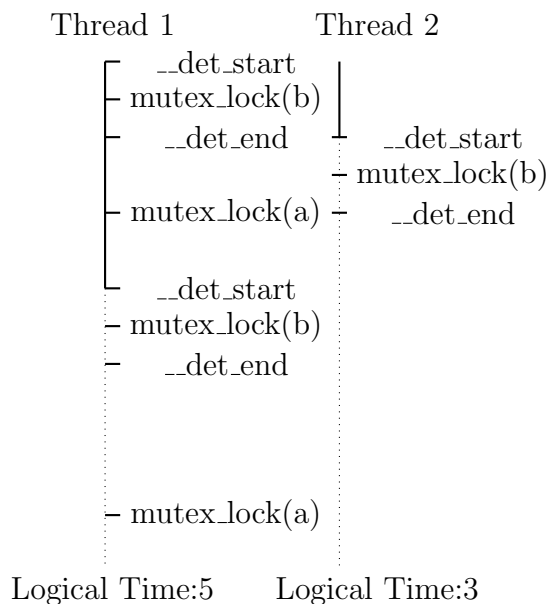
Figure 3.3: An example of deadlock

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:

producer   consumer

produceItem

__det_start   __det_start

mutex_lock   mutex_lock

__det_end   __det_end

putItem   getItem

mutex_unlock   mutex_unlock

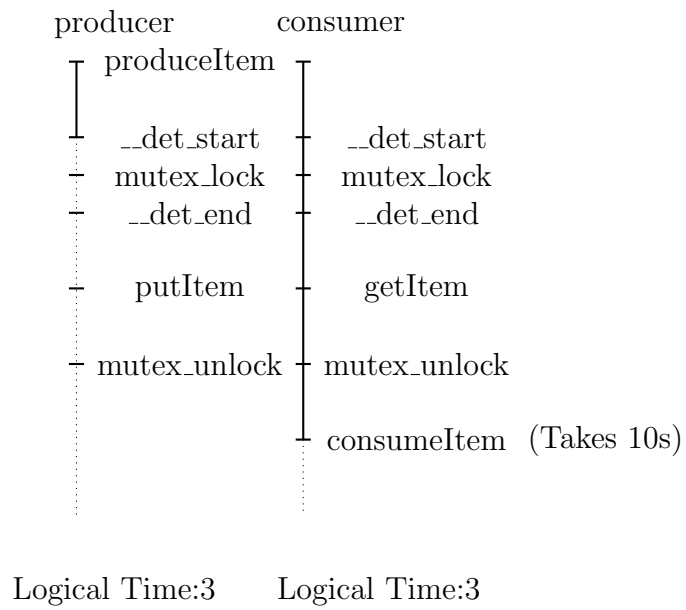consumeItem (Takes 10s)

Logical Time:3   Logical Time:3
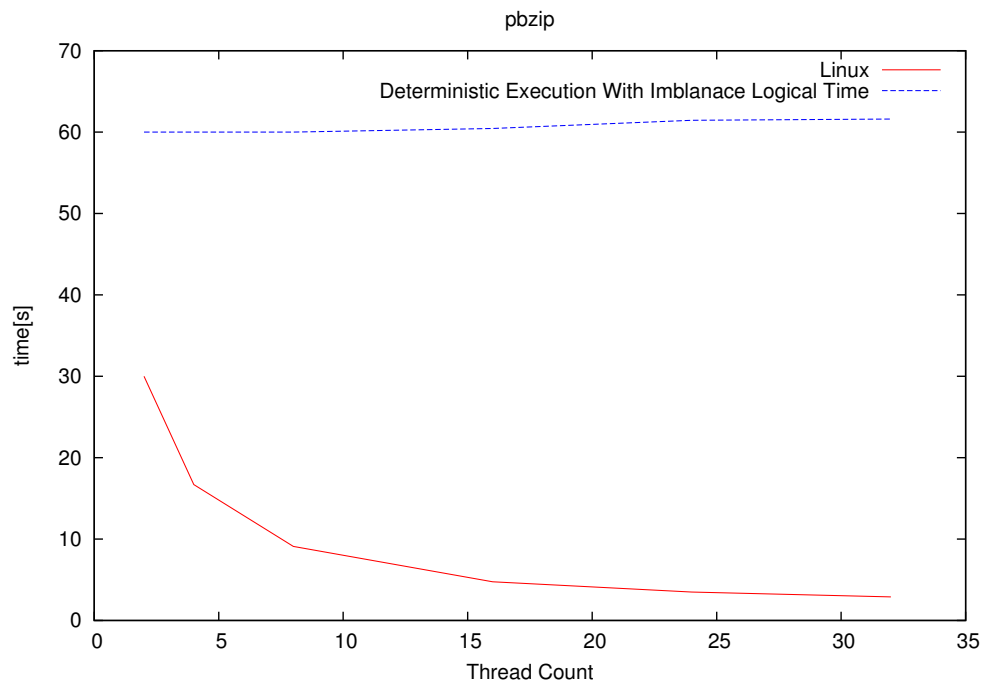
Figure 3.4: An example of logical time imbalance.

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
```

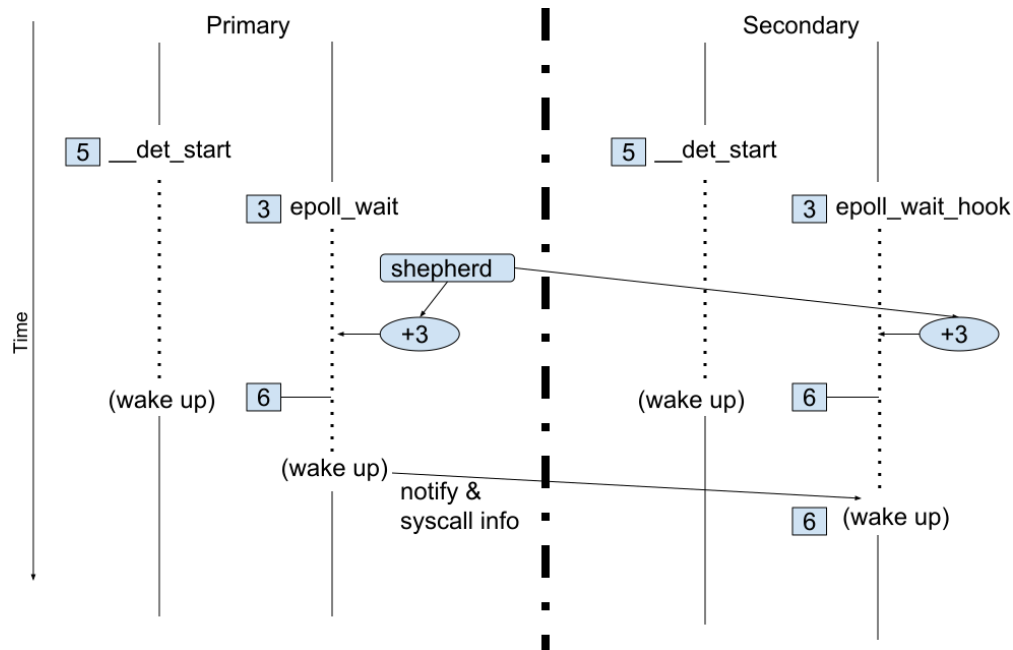Figure 3.7: An instrumented basic block in pbzip2 with dettick.

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

3.3.1   Deterministic Language Extension

Clik++ [**?**] 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 [**?**] 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   Deterministic Runtime

Weak Determinism

Weak Deterministic systems usually only target on making synchronization primitives to be deterministic. Kendo[**?**], Parrot[**?**] and Dthreads[**?**] 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. However they are limited to pthread only applications. Among these three, Kendo is the basis of our algorithm. However it relies on hardware counters to keep track of the program's progress in runtime, given the fact that hardware counters could be non-deterministic[**?**], we doubt the determinism of Kendo in our replication case.

Strong Determinism

Strong Deterministic systems aims to make every shared memory access to happen in a deterministic order. DMP[**?**] 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 DMP's overhead is too high due to massive trapping to shared memory accesses.

```
 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

3.3.3   Architectural Determinism

In [**?**], an architectural solution is proposed. It's a hardware layer between the CPU core and memory hierarchy, the goal is to track all the memory access and does versioning on the memory operations. By doing deterministic submission to the memory hierarchy, it ensures the determinism of the parallel execution. Although it's a promising solution which is totally transparent to the upper layer, it's not usable out of box in recent years.

3.3.4   Deterministic System For Replication

Chapter 4

Nigoki: Schedule Replication System

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, Figure 4.1 shows the relation between two different algorithms. Fortunately, our system is for inter-kernel replication, and Popcorn Linux provides a messaging layer with negatable latency. As a result having massive massages between replicas won't put too much overhead to the replication.

Figure 4.1: Trade off between two algorithms

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.

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). 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 **??** shows a simplified version of __det_start and __det_end in Schedule Replication.

Figure **??** shows an example of how Schedule Replication works in action.

4.2   Related Work

```
 1  void __det_start ()
 2  {
 3      if (is_secondary(current))
 4          wait_for_sync(current->seq,
 5              ns->seq, current->ft_pid);
 6      lock(ns->global_mutex);
 7      current->ft_det_state = FT_DET_ACTIVE;
 8  }
 9  void __det_end()
10  {
11      if (is_primary(current))
12          send_sync(c
13  2263-25 - Available: June 27, 2016 - Upstairs Unit
14
15  Leasing Office: 2249 Ulric Street, San Diego, CA 92111
16  Hours: 9:00am - 6:00pm Monday - Friday and 9:00am - 4:00pm Saturday
17
18  Large Studios with ALL utilities included
19
20  Non-Refundable Application Fee: $35.00 (18 years and older)urrent->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 4.2: Simplified implementation of system calls for schedule replication

Primary

Secondary

G=10
T=3

__det_start

mutex_lock

__det_end

G=10
T=2

__det_start

G=10
T=3

__det_start

G=10
T=2

__det_start

Time

G=11
T=4

(wake up)

mutex_lock

__det_end

(wake up)
mutex_lock
__det_end

G=12
T=3

G=11
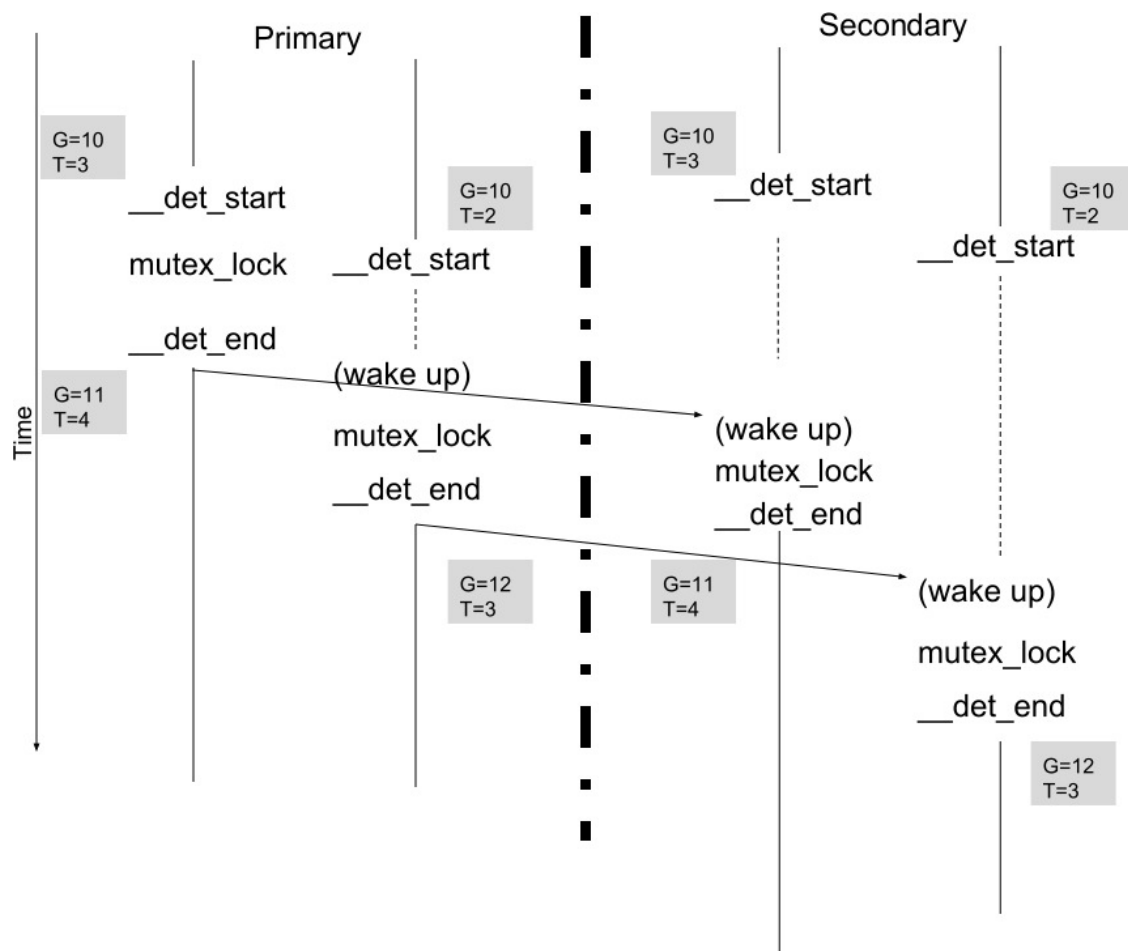T=4

(wake up)

mutex_lock

__det_end

G=12
T=3

Figure 4.3: An example of Schedule Replication

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 our implementation we only synchronize those system calls that are strongly related to network I/O: gettimeofday/time, poll, epoll_wait. We didn't implement select because it is relatively out-dated, modern network applications hardly use it, as a result it doesn't worth to put any effort into 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   Override 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.

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 function in it, and the namespace launching script will automatically set LD_PRELOAD 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 Override 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   Override 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. 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.

In our LD_PRELOAD library, we re-implemented pthread_cond_wait following the existing glibcś implementation, and wrapped every single 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.
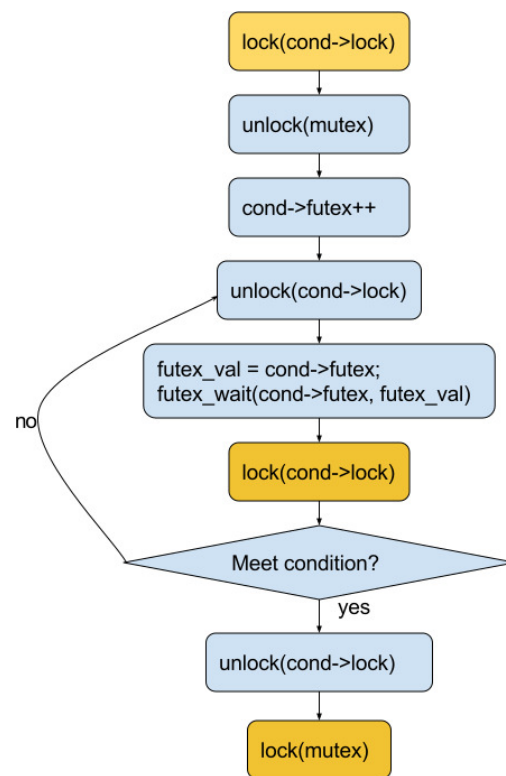
Figure 5.4: glibc pthread_cond_wait internal work flow

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 "preserved" 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 process's console output.

Chapter 6

Evaluation

In this chapter we will show some experiment results of our system,

6.1 Correctness Evaluation

6.1.1 Racey Benchmarks

6.2 PBZip2

6.2.1 Overhead Profiling

6.2.2 Results

6.3 Mongoose Webserver

6.3.1 Overhead Profiling

6.3.2 Results

6.4 Nginx Webserver

6.4.1 Overhead Profiling

6.4.2 Results

6.5 Redis Database Server

6.5.1 Overhead Profiling

6.5.2 Results

Chapter 7

Conclusion

7.1 Contributions

7.2 Future Work

7.2.1 Pre-Lock Synchronization

7.2.2 Arbitrary Number Replicas

7.3 Further Evaluation

Chapter 8

Bibliography