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.

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

1 Introduction **1**

2 Popcorn Linux Background **2**

2.1 Multikernel Boot 2

2.2 Inter-Kernel Messaging Layer 2

2.3 Popcorn Namespace 2

2.3.1 FT PID 2

2.4 Network Stack Replication 2

3 Deterministic Execution **3**

3.1 Logical Time Based Deterministic Scheduling 3

3.2 Balance the Logical Time 4

3.2.1 Execution Time Profiling 5

3.2.2 Tick Bumping for External Events 6

3.3 Eliminate Deadlocks 6

3.4 Related Work 8

4 Schedule Replication **9**

4.1 Execute-Follow Model 9

4.2 Implementation 9

5 Additional Runtime Support **10**

5.1 Synchronization Exclusion 10

5.2 Syscall Synchronization . . . 10
5.3 Modified Pthread Library . . . 10
5.4 Deterministic State Debugging Utilities . . . 10

6 Evaluation **11**
6.1 Correctness Evaluation . . . 12
6.1.1 Racey Benchmarks . . . 12
6.2 PBZip2 . . . 12
6.2.1 Overhead Profiling . . . 12
6.2.2 Results . . . 12
6.3 Mongoose Webserver . . . 12
6.3.1 Overhead Profiling . . . 12
6.3.2 Results . . . 12
6.4 Nginx Webserver . . . 12
6.4.1 Overhead Profiling . . . 12
6.4.2 Results . . . 12
6.5 Redis Database Server . . . 12
6.5.1 Overhead Profiling . . . 12
6.5.2 Results . . . 12

7 Conclusion **13**
7.1 Contributions . . . 13
7.2 Future Work . . . 13
7.2.1 Pre-Lock Synchronization . . . 13
7.3 Further Evaluation . . . 13

8 Bibliography **14**

List of Figures

3.1 Simplified version of deterministic system calls 4
3.2 An example of logical time imbalance. 5
3.3 An example of deadlock 7

List of Tables

1.1 The Graduate School wants captions above the tables. 1

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 FT PID

2.4 Network Stack Replication

Chapter 3

Deterministic Execution

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. Among all the deterministic systems, there is one type called "Weak Deterministic System". This type of systems assume 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, we implemented a set of system calls to control the application's scheduling at given points, which in turn controls the thread interleaving.

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. Our system provides three system calls for the applications to control the thread-interleaving:

- __det_start: Upon it is called, only the task holds the minimal logical time can proceed, if several tasks have the same logical time, the one who has the smallest PID number gets the turn. If the current thread is able to proceed, this thread will be marked as "in a deterministic section".
- __det_end: When is called, the system will increase the current thread's logical by 1, and marks it as "out of a deterministic section".
- __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.

Deterministic Logical Time

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

Figure 3.1: Simplified version of deterministic system calls

. 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 3.1 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 and pthread_spin_lock, so that the order of getting into critical sections is controlled under our deterministic scheduling.

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 3.2, we show

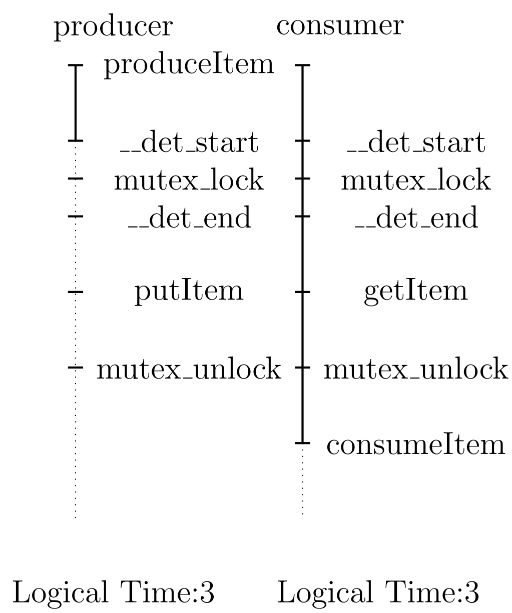

Figure 3.2: An example of logical time imbalance.

a particular execution point of the producer-consumer model, 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.

Generally, logical time imbalance can happen in two cases:

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

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. 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. 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 **??** shows an example of how Tick Shepherd works. 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.

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 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 means that 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

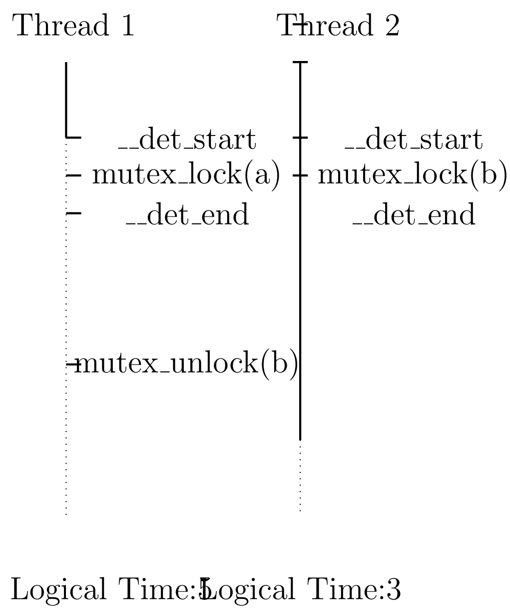

Figure 3.3: An example of deadlock

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. Simply doing so might break the determinism, we ensure the determinism by guaranteeing the following:

- 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 timing of getting out of a contended pthread_mutex_lock will be the same sequence. This is implemented by changing the priority queue inside futex_wait to a FIFO queue.
- 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 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.

3.4 Related Work

Deterministic systems have been studied for a long time. From the implementation perspective view, they can be categorized into 4 different genres: language level, runtime level, OS level and architectural level.

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.

Kendo[?], Parrot[?] and Dthreads[?] 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. Although Melchior can only make pthread to run deterministically in an automatically way, a developer is always free to use the runtime system calls to hand tune any type of parallel applications to make them deterministic. Among these three, Kendo uses the same deterministic scheduling policy as Melchior. 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 some cases. 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. We synchronization provided by the programming model, this could be unnecessary.

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.

Chapter 4

Schedule Replication

4.1 Execute-Follow Model

4.2 Implementation

Chapter 5

Additional Runtime Support

5.1 Synchronization Exclusion

5.2 Syscall Synchronization

5.3 Modified Pthread Library

5.4 Deterministic State Debugging Utilities

Chapter 6

Evaluation

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

7.3 Further Evaluation

Chapter 8

Bibliography