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

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 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. Among all the deterministic systems, there is one type called "Weak Deterministic System". Those systems assume the applications are 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 behaviour. By putting our system calls around the synchronization primitives, we are able to control the interleaving of those code sections.

3.1 Logical Time Based Deterministic Scheduling

Inspired by Kendo[**?**] and Conversion[**?**], this scheduling policy maintains a logical time for each task inside the 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.

. 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 it up. As long as the replicated ap-

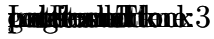

Figure 3.1: An example of logical time imbalance.

plication updates logical time in a same way on both primary and secondary, they will sure end up with the same thread interleaving.

3.2 Balance the Logical Time

Only increase 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 3.1, we show a particular execution point of the producer-consumer model that corresponds to the program shown in Figure **??**. 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 part, this waiting time is totally unnecessary. In a more complex application with more synchronization, this kind of waiting will break a parallel program into a serial program.

Logical time imbalance can happen in two cases:

- A task is running in the userspace for a long time.
- A task is sleeping in the kernel space for a long time.

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

3.2.1 Execution Time Profiling

3.2.2 Non-deterministic External Events

Some blocking system calls would put the caller into sleep. When a thread holding a token reaches such a system call, the token might be on this thread for a long time. Especially for system calls like epoll_wait, poll and accept, 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. In order to let the token passing keep going with those blocking system calls, a "Tick Shepherd" is implemented to dynamically bump the logical time of the threads that are sleeping on external events. The 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 its logical time and send the delta to the replica. In the meanwhile the corresponding system call on the replica will be blocked at the beginning, and bump 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.

Figure 3.2: 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 along with the epoll_waitś output 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.

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

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