Melchior: A Compiler Assisted Deterministic System

Yuzhong Wen[*]
Virginia Tech
wyz2014@vt.edu

ABSTRACT

The non-determinism execution order on multiprocessing system is a double-edge, while the maximum performance can be gained from this, the users and the developers are sometimes suffered from non-repeatable results for the same input. Especially for developers, debugging potential data races or threading related bugs in this case could be a disaster.

However with a deterministic system, it can be guaranteed that for a given input, a multi-threaded program is always executed in a same thread scheduling no matter how many times it runs. In this paper we present Melchior, a deterministic system that guarantees deterministic scheduling for a group of threads/processes. Melchior consists of a compiler part which instruments the program, a user-space utility to launch the deterministic namespace, and a patch to Linux kernel to control the scheduling inside the original Linux scheduler. The goal of Melchior is to control the sequence of inter-thread/process communication, in terms of thread synchronization, share memory access and system calls.

In this paper we also discussed the scalability of a deterministic system, and with Melchior's compiler support, we are able to mitigate this problem and achieve a relatively high scalability, the runtime overhead with big number of threads (up to 64) is much less than most existing deterministic systems. Among all the system level deterministic systems, Melchior is the fastest. We evaluated the implementation with a traditional parallel workload, pbzip2.

Keywords

Deterministic Multithreading; Multicore; Determinism

1. INTRODUCTION

1.1 Background

The non-determinism execution order on a multi-core system is a double-edge, while the maximum performance can be gained from this, the users and the developers are sometimes suffered from non-repeatable results for the same input. This non-determinism comes from the parallel execution of different threads on different cores. Given the fact that each core's progress might be subject to change due to the OS scheduler, the time which each thread reaches the shared memory access would be different on every execution. While this non-determinism behaviour of multi-threaded programs is widely accepted by programmers, those developers are suffering from debugging those non-deterministic thread interleavings.

A workaround for debugging non-deterministic multi-threaded programs is via a record/replay system. A typical record/replay system runs the program in a specific runtime and logs the execution of the program, and during the replay phase, a debugger can replay the previous execution in a serial manner, which is easier to debug[16]. Also there're ways to replay the program in a parallel manner by only recording the schedule of the record run, so that on the replay run the program can be scheduled according to the previous schedule log[7]. However, an important fact is that the record run for a record/replay system is still not deterministic.

As a result, the need for a deterministic system still exists. Researchers from different fields are trying to tame the non-determinism in language level[5][3][11], runtime level[1][6][12][2][8][14] even architectural level[15]. In fact, with determinism guaranteed, the programs' execution can be easily understood during the runtime, regardless different thread interleavings. It is also suggested that parallel programming should born to be deterministic[4].

1.2 Melchior's Determinism

For multi-threaded programs, an observation is that as long as the threads don't communicate, the execution is sure to be deterministic[8]. 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[6][12][14].

Melchior's 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. In this way we have a more flexible deterministic solution for programmers to control the behaviour of their parallel programs. Unlike other runtime deterministic systems, Melchior's runtime does not directly trap pthread primitives, but provides two system calls for programmer to define a deterministic area. While Melchior's compiler framework can automatically instrument pthread primitives, a developer can also manually use those two system calls to hand-tune their applications. Figure 1 shows the use of the system calls on a simple producer-consumer program. Line 5-7 and

[*]

```
1  void producer() {
2    while (running){
3      item = generate_item();
4
5      syscall(__NR_det_start);
6      pthread_mutex_lock(mutex);
7      syscall(__NR_det_end);
8
9      putItem(queue, item);
10
11     pthread_mutex_unlock(mutex);
12   }
13 }
14
15 void consumer() {
16   while (running){
17     syscall(__NR_det_start);
18     pthread_mutex_lock(mutex);
19     syscall(__NR_det_end);
20
21     item = getItem(queue);
22
23     pthread_mutex_unlock(mutex);
24
25     consume_item(item);
26   }
27 }
```

Figure 1: An example use of Melchior's deterministic hints

line 17-19 are two deterministic areas. Only the thread has the execution token can proceed at *syscall(__NR_det_start)*, for other theads who also reach *syscall(__NR_det_start)*, Melchior's runtime will put them into sleep until one of them has the token. In this case the order of executing pthread_mutex_lock is controlled by Melchior runtime in a deterministic way. As a result, putItem and getItem are also executed in a deterministic order since they are protected by a same mutex.

1.3 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++ [11] 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 [3] 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[14], Parrot[6] and Dthreads[12] 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[17], we doubt the determinism of Kendo in some cases. DMP[8] 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 [15], 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.

1.4 Contribution

This paper makes following contributions:

- We present Melchior, a deterministic system with a runtime and a compiler framework that can make any pthread based program to run in a deterministic way transparently. Also supports fork based programs with manually annotation.
- We address the challenge of achieving high scalability in a deterministic system, and proposed a solution to mitigate the problem with our compiler framework.
- We evaluated Melchior's correctness with several micro-benchmarks, we show that with the same input, Melchior can always enforces the same output for those micro-benchmarks.
- We evaluated Melchior's performance with pbzip, where we achieve a relatively low overhead(fromit 17% to 170% for different number of threads), among all the deterministic systems that presented pbzip results, we are the 2nd fastest.

2. DETERMINISTIC SCHEDULING

2.1 Deterministic Logical Time

Inspired by Kendo[14] and Conversion[13], Melchior's maintains a "logical time" for each task inside the Melchior's runtime. Only the task holds the minimal logical time can have the token, if several tasks have the same logical time, the one who has the biggest PID number gets the token. As shown in Figure 1, the logical time increases by 1 when *syscall(__NR_det_end)* is called. Suppose the logical time for all the thread was 0 initially, after *syscall(__NR_det_end)* is called by a thread who has the token, its logical time will be 1. At this point the token will be released from this thread and will be handed over to another thread, the one who gets the token will be waken up from previous *syscall(__NR_det_start)* and proceeds to its deterministic area. In this algorithm, the logical time is increased deterministically, as a result, the token is passed in a deterministic way. We modelled a simple multi-producer-multi-consumer program along with this scheduling algorithm in TLA+. In

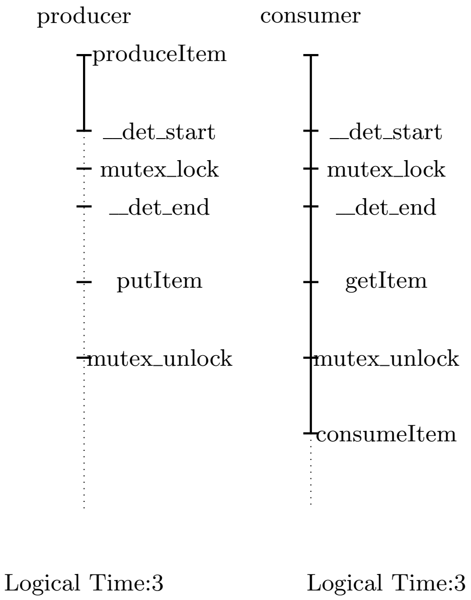

Figure 2: An example of logical time imbalance. In this case the consumer has the token and already released the mutex, however consumeItem is taking too much time before it reaches __det_end again. At this time producer has to wait until consumeItem is finished and logical time of consumer becomes 4.

the model, consumers generate some output according to the item they get in parallel. With variants defined, we observed that the sequence of the consumers' output never changes. Which proves the correctness of this algorithm.

2.2 Logical Time Imbalance

However, after this algorithm simply was implemented, we found that the program's performance doesn't scale as the thread count increases, in other words, the scalability was very bad. The root cause is that we only increase the logical time by 1 at *syscall(__NR_det_end)*. With an example we show how this could break the scalability and how to mitigate this problem.

In Figure 2, we show a particular execution point of the producer-consumer model that corresponds to the program shown in Figure 1. 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.

From this case, we can make a conclusion: the logical time should precisely reflect the progress of the thread, only increasing the logical time by 1 at the end of synchronization is not enough. For this purpose, we add another system call just for adding logical time to the runtime. In Figure 3 we show that by increasing the logical time according to the execution time of consumeItem, the producer can proceed. Since this happens out of the deterministic area, as long as the logical time is increased in a deterministic manner, it won't break the determinism.

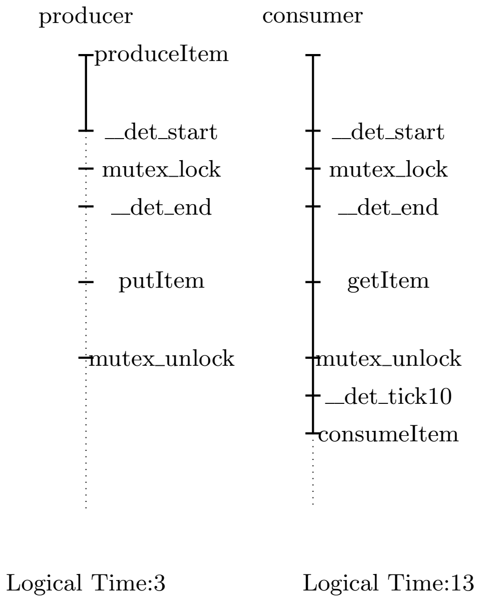

Figure 3: A solution for the logical time imbalance. Here we add the logical time of consumer by 10 before it reaches consumeItem, so that the producer can proceed without waiting for the consumer.

3. MELCHIOR

In this section we present the detail design of Melchior. Melchior's runtime provides 2 system calls to define a deterministic area and 1 system call to bump the logical time. Melchior's compiler framework automatically instrument a program's so that every pthread synchronization is wrapped to be a deterministic area, and it can also find the best place and value to bump the logical time, achieving maximum scalability.

3.1 Linux-Based Runtime Implementation

The runtime is implemented purely in Linux kernel version 3.2.14. In order to not mess up some other running tasks in the OS, Melchior comes with a Linux namespace. Only the processes inside this namespace will affected by the deterministic scheduler. The namespace maintains a list of all the running tasks inside it, the list is ordered by PID.

Once the __det_start is called by a thread, the kernel will mark this thread as in a deterministic area, and check if the thread has the token, otherwise it removes the thread out of the current CPU run queue. When __det_end is called, the logical time will be increased by 1 and mark this thread as not in a deterministic area. Every time the logical time is updated, the runtime will put the token to the one that has the minimal logical time. If the thread that has the token is sleeping due to waiting on a token, the kernel will wake it up and put it back to the CPU run queue.

3.2 LLVM-Based Compiler Framework

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

Figure 4: An instrumented basic block in pbzip2. This is a part of the consumer of pbzip2, here it takes an input file block and compress the content. According to previous recorded data this part should be time consuming, therefore a system call is inserted to bump the logical time of this thread at line 10.

3.2.1 Pthread Annotation Pass

This pass is very straightforward, it analysis the program's source code and puts __det_start and _det_end around the pthread sychronizations. With this pass all the pthread-based program can be made into deterministic-ready automatically. However a programmer can also manually put those two system calls into a program that has different threading model, even applicable for fork-based multi-process programs.

3.2.2 Profile Pass

As mentioned in Section 2.2, we need to bump the logical time at certain places of the program, and it has to reflect the actual execution time of the program. In order to get the execution time of each section of a program, we make a profile pass to collect the execution time of each basic block, and then output the profile result to a file.

bzBuffToBuffCompress

3.2.3 Logical Time Pass

After the program finished one profile run with the instrumentation of profile pass, the profiler will generate a file that encodes the execution time of each basic block. The logical time pass will take the profile data file as input, and compile the program again. 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. Figure 4 shows an example of instrument basic block in LLVM-IR. Since the value of logical time bumping is determined during the compilation time and won't change , the runtime binary is sure to be deterministic.

4. EVALUATION

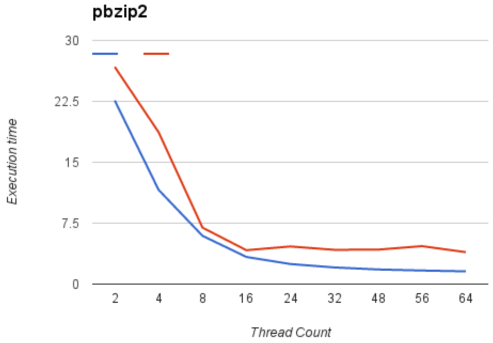

Figure 5: Performance benchmark for pbzip2. Red line is Melchior, blue line is non-deterministic run. With logical time balancing, the scalability is almost close to the non-deterministic run.

The goal of our evaluation is to evaluate both the performance and correctness of our system. The evaluation was done on a server with 4 AMD Opteron 6376 processors, 64 cores in total, which is sufficient for the scalability evaluation.

4.1 Correctness

We evaluated Melchior with a variant of racey [10], which protects the shared memory access with mutexes. In our test, racey produces the same result for 2000 runs. And with some manually annotation work, two variants of racey that are implemented with fork instead of pthread, also passed the test for 2000 runs.

4.2 Performance

The workload we chose is pbzip2 [9]. It utilizes a typical producer-consumer model with selectable numbers of threads which is perfectly suitable for testing the scalability of our system. We compress a 177MB file with different number of threads. The block size for pbzip2 is set to be 15MB.

Without logical time balancing involved, we didn't see any scalability. Regardless the number of threads, the deterministic execution always performs as the performance of one single thread.

Figure 5 shows the result with logical time balancing. The scalability is pretty well with our instrumentation. The overhead of the runs varies from 17% to 170%, which is reasonable for a deterministic system. In DMP [8] it was shown that the overhead of pbzip2 is 3x under 8 threads, while in Parrot [6] their overhead is under 50%. Melchior is the 2nd fastest by far.

5. CONCLUSION AND FUTURE WORK

We present Melchior, a runtime plus compiler framework based deterministic system. Melchior's runtime allows a programmer to handtune the program to be deterministic, while the compiler framework can automate this process for pthread based programs. Also we address the problem of the logical time imbalance and propose a compiler-based approach to mitigate the problem, where we achieved a relatively low overhead that is pretty much enough for a production system.

As shown in the benchmark, we still have some space for improvement. One could be improve the logical time balancing in terms of position and value. The currently implementation is just heuristically add logical time bumping at the end of each basic block. This approach is not the best solution because: 1. The best place for inserting the logical time bumping should be before the instruction that is time consuming as shown in Figure 3, bumping the logical time at the end of basic block works for pbzip2, but we are not sure if it also works for other programs. 2. Calling the logical time bumping system call too frequently may leads to some unnecessary overhead of context switch. We could develop an approach to minimize the number of calls by a better analysis of the execution time.

6. REFERENCES

[1] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM*, 55(5).

[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 53–64. ACM, 2010.

[3] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for c/c++. In *ACM Sigplan Notices*, volume 44, pages 81–96. ACM, 2009.

[4] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 4–4, 2009.

[5] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. *ACM Sigplan Notices*, 44(10):97–116, 2009.

[6] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*.

[7] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *OSDI*, volume 10, page 10, 2010.

[8] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 85–96. ACM, 2009.

[9] J. Gilchrist. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, volume 16, pages 559–564, 2004.

[10] M. Hill and M. X. Racey. A stress test for deterministic execution.

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

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

[13] T. Merrifield and J. Eriksson. Increasing concurrency in deterministic runtimes with conversion.

[14] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108, 2009.

[15] C. Segulja and T. 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.

[16] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: parallelizing sequential logging and replay. *ACM Transactions on Computer Systems (TOCS)*, 30(1):3, 2012.

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