Semantic-Aware Synchronization Determinism and Beyond

Qi Zhao*

North Carolina State University, Raleigh NC 27606, USA
qzhao6@ncsu.edu

1 Background

Multi-threaded programs are often executed non-deterministically. As a result, threads can interleave in many different ways. This is a major obstacle in writing correct multi-threaded programs, as a different thread interleaving can lead to a different output of a program, even with the same input. It also complicates debugging and testing of multi-threaded programs, as buggy thread interleaving may be difficult to reproduce. Because of this nondeterminism, concurrency bugs are referred to as "heisenbugs" [3], a pun on the Heisenberg's uncertainty principle in quantum physics.

Deterministic multithreading (DMT) systems aim to eliminate this nondeterminism in multi-threaded programs. They try to ensure that a program's output solely depends on the program's input, regardless of how the program is coded or if they originally contain heisenbugs or not. What DMT systems achieve can be divided into two tasks: (1) synchronization determinism, which means for any given input, the sequence of synchronizations will remain the same across executions, and (2) memory-access determinism, which guarantees all shared memory reads will return the same results across executions. Previous work [4, 5] has shown that memory-access determinism can be implemented on top of synchronization determinism without extra serialization. As a result, we keep our focus on reducing the overhead of synchronization determinism.

Although all DMT systems ensure the same schedule for the same input, some lacks schedule stability. That is, the schedules they enforce could be very different for different inputs. Concerning that buggy interleaving may be missed during testing due to this, researches proposed stable DMT systems that map similar inputs to the same schedule [1, 2, 6]. Among them, Parrot [1] schedule synchronizations using round-robin policy to achieve good schedule stability. This strategy works well when workloads and the number of synchronizations is distributed evenly among multiple threads, but it will incur high overhead otherwise. As a supplement, they also introduced an annotation called *soft barriers* to adjust scheduling while round-robin policy incurs high overhead. Developers can place these annotations before core computation code regions, to make sure these computations are executed in parallel. However, these annotations only address some sources of overhead and are not always easy to insert. Once inserted, they become part of the code that needs to be maintained.

In short, it's important for a DMT system to achieve stability and efficiency without the need of manual annotation.

* Advised by Guoliang Jin

Table 1. QiThread policies and the type of synchronizations they cover.

Name	Type of Sync.	Brief summary of the policy
BoostBlocked	Order enforcement	Prioritize threads just woken up from blocked state
CreateAll	Thread creation	Schedule consecutive thread creation all at once
CSWhole	Mutual exclusion	Schedule critical section as a whole
WakeAMAP	Order enforcement	Wait till we can wake up multiple blocked threads together
BranchedWake	Order enforcement	Insert dummy synchronization to rebalance branches

2 QiThread

In QiThread, we designed scheduling strategies that deviate from round-robin policy when certain synchronization sequences are observed to achieve low overhead without manual annotations. We summarized five synchronization sequence patterns, listed in Table 1, where applying alternative scheduling can have a good chance of reducing overhead. For each sequence, QiThread designed policies on how to schedule each thread regarding the sequence. For BoostBlocked policy, threads that are just woken up from the blocked state are scheduled first before other threads. In CreateAll, when a thread call `pthread_create` in a loop, all of them are scheduled together. When CSWhole is applied, a critical section is scheduled together as a whole. For WakeAMAP, the scheduler counts the number of threads blocked on a particular `pthread_cond_wait` variable, or semaphore, and keeps scheduling the thread that can wake them up, until all of the blocked threads are awake. Lastly for BranchedWake, if we found that some threads skip a simple branch that contains a `pthread_cond_signal` or a `sem_post`, we insert a dummy synchronization to those threads to restore the balance across threads. These policies cover different types of synchronizations. We have evaluated them on 108 programs from multiple benchmark suite and real-world applications. The results shown that, when combined together, they can achieve a similar or sometimes even better performance than Parrot with manual soft barriers.

3 Future Work

While QiThread relies solely on the synchronization operation patterns to guide its scheduling, a high-level analysis may be required for programs with more complex synchronization patterns. For future work, we propose an approach that analyzes high-level thread role and understand the data dependency relationship between threads. For example, some threads in a program may assume a producer role, while others act as consumers. Then the deterministic scheduler can decide which thread should have higher priority in executing the next synchronization. The system includes a tool to identify thread role with LLVM-based program analysis, and a deterministic synchronization scheduler.

References

1. Cui, H., Simsa, J., Lin, Y.H., Li, H., Blum, B., Xu, X., Yang, J., Gibson, G.A., Bryant, R.E.: PARROT: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In: SOSP '13. pp. 388–405. SOSP '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2517349.2522735
2. Cui, H., Wu, J., Tsai, C.c., Yang, J.: Stable Deterministic Multithreading Through Schedule Memoization. In: Proc. 9th USENIX Conf. Oper. Syst. Des. Implement. pp. 207–221. OSDI'10, USENIX Association, Berkeley, CA, USA (2010), http://dl.acm.org/citation.cfm?id=1924943.1924958
3. Gray, J.: Why do computers stop and what can be done about it. Symp. Reliab. Distrib. Softw. database Syst. **3**(2), 3–12 (1986). https://doi.org/10.1.1.59.6561
4. Lu, K., Zhou, X., Bergan, T., Wang, X.: Efficient deterministic multithreading without global barriers. In: Proc. 19th ACM SIGPLAN Symp. Princ. Pract. parallel Program. - PPoPP '14. pp. 287–300. PPoPP '14, ACM Press, New York, New York, USA (2014). https://doi.org/10.1145/2555243.2555252
5. Merrifield, T., Eriksson, J.: CONVERSION : Multi-Version Concurrency Control for Main Memory Segments. EuroSys pp. 127–140 (2013). https://doi.org/10.1145/2465351.2465365
6. Zhao, Q., Qiu, Z., Jin, G.: Semantics-aware scheduling policies for synchronization determinism. In: Proc. 24th Symp. Princ. Pract. Parallel Program. - PPoPP '19. pp. 242–256. No. 1, ACM Press, New York, New York, USA (2019). https://doi.org/10.1145/3293883.3295731