RaceFree: An Efficient Multi-Threading Model for Determinism

Kai Lu Xu Zhou Xiaoping Wang Wenzhe Zhang Gen Li
National University of Defense Technology
{kailu, zhouxu, xiaopingwang, zhangwenzhe, ligen}@nudt.edu.cn

Abstract

Current deterministic systems generally incur large overhead due to the difficulty of detecting and eliminating data races. This paper presents RaceFree, a novel multi-threading runtime that adopts a *relaxed deterministic model* to provide a data-race-free environment for parallel programs. This model cuts off unnecessary shared-memory communication by isolating threads in separated memories, which eliminates direct data races. Meanwhile, we leverage the happen-before relation defined by applications themselves as one-way communication pipes to perform necessary thread communication. Shared-memory communication is transparently converted to message-passing style communication by our Memory Modification Propagation (MMP) mechanism, which propagates local memory modifications to other threads through the happen-before relation pipes. The overhead of RaceFree is 67.2% according to our tests on parallel benchmarks.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging Aids; D.4.1 [*Operating Systems*]: Process Management—Threads

Keywords deterministic multi-threading, debug, synchronization, relaxed deterministic model, happen-before

1. Introduction

Parallel programs of shared-memory multi-threading are inherently nondeterministic due to the unpredictable thread interleavings, which brings difficulties to software debugging, testing and fault tolerance [1, 2]. Current techniques that could solve the nondeterministic problem contain *Deterministic Replaying* [6–9] and *Deterministic Multi-Threading* (DMT) [1–5]. These approaches apply different methods to eliminate the nondeterministic sources. However, due to the difficulty of detecting and eliminating the nondeterministic data races, current deterministic approaches are generally low-efficiency.

Thread concurrency exposes two kinds of nondeterminism to user programs: data race and synchronization race. In shared-memory multi-threading, threads could communicate with each other by directly reading and writing shared memories. The unprotected shared memory accesses may incur data race, which is the most common nondeterminism of parallel programs. Eliminating data races is difficult as they may occur at any memory access. In order to eliminate data race, we have to check every memory access to make sure the memory access will not conflict with another concurrent memory access. Compared with data race, synchronization race is much easier to handle. The reason is that synchronization race could only occur at well-defined synchronizations, which are (1) explicit in user programs, and (2) much fewer than memory accesses.

1.1 Relaxed Determinism

To address the performance problem of current deterministic systems, we introduce a novel threading model: *relaxed deterministic model*. The relaxed deterministic model provides a data-race-free multi-threading environment, in which only synchronization race could affect the determinism of parallel programs. Since this model does not guarantee strong determinism unless we ensure a deterministic synchronization order, the deterministic semantics this model promises is called *relaxed determinism*.

Relaxed determinism could be easily changed to strong determinism if the synchronization order is guaranteed to be deterministic. We deliberately leave this nondeterminism to other techniques (e.g., deterministic replay [6–8] and weak determinism [2]) to provide a spectrum of determinism choices for different application scenarios, as discussed in Section 1.4 .

1.2 Design

Data race is considered as unnecessary shared-memory communication between threads. The basic design of our relaxed deterministic model is to identify necessary communication from unnecessary communication, and cut off all the unnecessary shared-memory communication. Since the programmer knows exactly whether a communication is necessary or not, we leverage what the programmer writes in the codes as a clue—user-inserted synchronizations. We assume that any necessary thread communication is noticed and thus synchronized by the programmers. Hence, we could use the happen-before relation [10] defined by synchronizations to distinguish necessary communication and unnecessary communication. As shown in Figure 1, if two memory accesses do not have a happen-before relation, they are treated as unnecessary communication. Otherwise, they are necessary communication, and they should communicate in their original way. To preserve the original communication semantics, we ensure the following semantics invariant for our model:

A Read should always see the value of a Write if it is the last Write that happens before the Read.

To cut off unnecessary thread communication, we isolate threads in separated shared memories. However, this could also block the necessary communication. To address this problem, we propagate memory modifications among threads to simulate the original nec-

PPoPP'13, February 23–27, 2013, Shenzhen, China.
ACM 978-1-4503-1922-5/13/02.

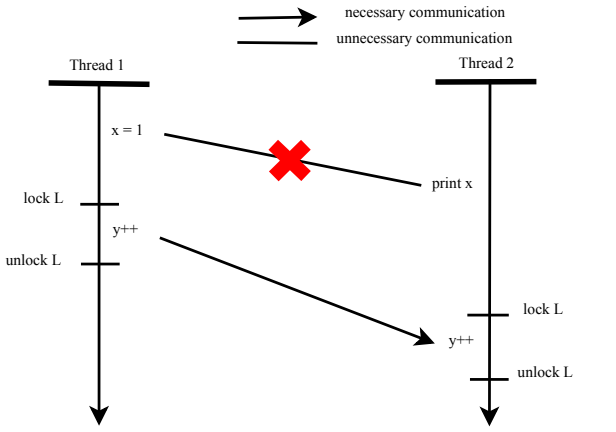

Figure 1. Shared-memory communication in relaxed deterministic model. Unnecessary communication is cut off while necessary communication is preserved.

essary communication. First, we interpose the synchronizations to capture the happen-before relation of the program. Then we perform necessary thread communication according to the happen-before relation, which is converted to one-way communication pipes between threads. Shared-memory communication could be converted to message-passing style communication by our Memory Modification Propagation (MMP) mechanism. MMP cuts thread execution into slices using synchronizations, collects the local memory modifications using the page protection mechanism, and propagates the modifications to other threads at the synchronization boundaries through the happen-before pipes. By this means, the original programs could run correctly and transparently on our model. Meanwhile, we constrain the nondeterministic data races to the boundaries of user-defined synchronizations. The characteristic of relaxed deterministic model is that it does not introduce extra synchronization schemes, thus threads could synchronize in the same way as their original design.

1.3 Results

We implemented a runtime system RaceFree to evaluate the relaxed deterministic model. We tested RaceFree in two aspects: determinism and performance. Since RaceFree only ensures relaxed determinism, we used the deterministic-replay approach to ensure a deterministic synchronization order for RaceFree. We checked whether this modified system could ensure determinism for parallel programs. Performance was evaluated by comparing RaceFree with the nondeterministic pthread library. The results on *racey* and parallel benchmarks (from Splash-2, Parsec and Phoenix) show that RaceFree could ensure relaxed determinism and the average overhead is 67.2% compared to pthread.

1.4 Application Scenario

Relaxed determinism that RaceFree ensures could be used to implement a spectrum of determinism for different applications. Here we discuss several application scenarios of RaceFree.

Implementing DMT. A DMT system is useful in many domains, such as debugging, testing and fault-tolerance [1–4]. Since only the synchronization order affects determinism, RaceFree could be upgraded to a DMT system with little effort. We note that *weak determinism* only ensures the deterministic synchronization order of a program [2], which is exactly complementary to our system. Hence, to build a DMT system, we only need to change the implementation of synchronizations in RaceFree, and use logical time to arbitrate the lock contentions [2].

Facilitating Deterministic Replay. Deterministic replay could be used for parallel debugging and intrusion analysis [11]. The most difficult part of deterministic replay is to identify and record data races, which often leads to complicated system design, large recording overhead and large disk-space overhead [9, 11]. With RaceFree, deterministic replay is simple and efficient: we only need to interpose synchronizations to record their execution order.

Reproducing Bugs. In parallel debugging, it is acceptable to reproduce a hidden bug in a few re-execution tries [6]. The data-race-free environment can be leveraged to implement such a system. As only synchronization race exists, the nondeterministic interleaving space is reduced. Hence, RaceFree is able to increase the probability of reproducing bug. Moreover, the reproducing of nondeterministic bugs could be directed by calculating the detailed scheduling information from the sketchy information of the original run (e.g., program outputs) to further increase the reproducing probability [6, 7].

Acknowledgments

This work is partially supported by National High-tech R&D Program of China (863 Program) under Grants 2012AA010901, 2012AA01A301, NCET, and National Science Foundation (NSF) China 61272142, 61103082, 61003075, 61170261, 61103193.

References

[1] D. Joseph, L. Brandon, C. Luis, and O. Mark, "DMP: deterministic shared memory multiprocessing," in Proceeding of the 14th international conference on Architectural support for programming languages and operating systems Washington, DC, USA: ACM, 2009.

[2] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," in Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, 2009, pp. 97-108.

[3] X. Zhou, K. Lu, X. Wang, and X. Li, "Exploiting parallelism in deterministic shared memory multiprocessing," J.ParallelDistrib.Comput., pp. 72(2012)716-727, 2012.

[4] B. Tom, A. Owen, D. Joseph, C. Luis, and G. Dan, "CoreDet: a compiler and runtime system for deterministic multithreaded execution," in Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 53-64.

[5] T. Liu, C. Curtsinger, and E. D. Berger, "DTHREADS: Efficient Deterministic Multithreading," in Proceedings of the 22nd ACM Symposium on Operating Systems Principles, 2011.

[6] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: probabilistic replay with execution sketching on multiprocessors," in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 177-192.

[7] G. Altekar and I. Stoica, "ODR: output-deterministic replay for multicore debugging," in Proceedings of the 22nd ACM Symposium on Operating Systems Principles, 2009.

[8] D. Subhraveti and J. Nieh, "Record and transplay: partial checkpointing for replay debugging across heterogeneous systems," 2011, pp. 109-120.

[9] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization, 2010, pp. 2-11.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, vol. 21, pp. 558-565, 1978.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: enabling intrusion analysis through virtual-machine logging and replay," ACM SIGOPS Operating Systems Review, vol. 36, pp. 211-224, 2002.