OPR: Deterministic Group Replay for One-Sided Communication

Xuehai Qian Koushik Sen
University of California, Berkeley
{xuehaiq,ksen}@cs.berkeley.edu

Paul Hargrove Costin Iancu
Lawrence Berkeley National Laboratory
{phhargrove,cciancu}@lbl.gov

The ability to reproduce a parallel execution is desirable for debugging and program reliability purposes. In debugging (13), the programmer needs to manually step back in time, while for resilience (6) this is automatically performed by the the application upon failure. To be useful, replay has to faithfully reproduce the original execution. For parallel programs the main challenge is inferring and maintaining the order of conflicting operations (data races). Deterministic record and replay (R&R) techniques have been developed for multithreaded shared memory programs (5), as well as distributed memory programs (14). Our main interest is techniques for large scale scientific (3; 4) programming models.

Shared memory R&R techniques use either information about thread scheduling (5) by tracking synchronization APIs, or log the memory accessed within each thread. In distributed memory, R&R techniques for MPI (14) have been developed with emphasis on scalability. They track two-sided `MPI_Send/MPI_Recv` operations and ignore local memory accesses. None of the existing approaches can provide deterministic R&R for the new class of modern distributed programming models (MPI-3 RMA (4)) and Global Address Space (UPC (3), Co-Array Fortran, Chapel, X10, OpenSHMEM which advocate one-sided communication abstractions.

In this paper, we present the first general tool, *OPR* (**O**ne-sided communication **P**artial **R**ecord and Replay) to support deterministic R&R for one-sided communication. The tool allows users to select a small set of threads of interest from a large scale application. It tracks their execution and upon demand it can deterministically replay the selected set of threads. As all other threads are not executed during the partial replay, the tool eases debugging experience and relieves users from monitoring all concurrent events from potentially tens of thousands of threads. OPR also makes it possible to debug a large-scale execution on a smaller (local) machine. Furthermore, partial replay is intrinsic to the scalability of resilience techniques (6) using uncoordinated or quasi-synchronous checkpointing and recovery.

Our OPR prototype is built for the Unified Parallel C (1) programming language. This is a typical PGAS (Partitioned Global Address Space) language whose memory consistency model allows for reordering of operations and therefore nondeterministic execution. Memory can be accessed either with load/store instructions or using one-sided communication (Put/Get). The challenge is to build a hybrid scalable mechanism able to infer the order of these disjoint multiple types of operations.

State-of-the-art deterministic R&R for shared memory programming (10; 12) handles load/store operations using value logging (referred to as data-replay (8; 12)). Determinism is attained by maintaining a shadow memory and comparing its contents against the program execution. In OPR, we use a similar approach to detect thread state changes due to remote direct loads/stores in record phase and log values at certain points.

Although the data-replay based approach enables replay in isolation, it does not provide sufficient insight on how communication happened between threads. To eliminate this drawback, we employ a *hybrid R&R scheme*. The data-replay which ensures correctness is complemented with order-replay (8) to infer inter-thread communication based on value matching. In the record phase, OPR runs a simplified and scalable vector clock algorithm only among the monitored threads to get an approximation of event orders of accesses to global memory. In the replay phase, OPR enforces the same event order and infers the communication by matching values of local writes and remote reads (Gets) (in the value log of remote threads). By combining an approximate order with matching the values in the logs, we provide scalability as well as allowing for non-atomic monitoring and recording of load/store and Put/Get operations. To the best of our knowledge, OPR is the first scheme that uses this hybrid approach.

The evaluation is conducted on Edison, a Cray XC30 supercomputer at NERSC. We evaluate OPR using eight NAS Parallel Benchmarks (2) (BT, CG, EP, FT, IS, LU, MG, SP), two applications using work stealing from the UPC Task Library (9) (fib, nqueens), three applications in the UPC test suite (guppie, laplace, mcop) and Unbalanced Tree Search (UTS) (11). In addition we evaluate a large scale production application performing Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly (Meraculous) (7). We focus on recording overhead and ensure that the output and the orders are right. Since a small number of threads are partially replayed, the threads can be replayed efficiently without any noticeable performance degradation. Therefore, in our experimental evaluation we only check replay fidelity and we do not focus on measurement of replay overhead. All applications are first executed on about 40 nodes (1,024 cores/threads) of Edison and we monitor and replay threads that can be contained on single node (two up to 16 cores/threads). We see that OPR incurs overhead from 1.3x $\sim$ 29x among all applications and different R_Set sizes (2,4,8,16 threads), when running the original program on 1,024 cores. Such overhead is moderate and acceptable for a software-only R&R scheme used for debugging. As discussed in Section **??**, we believe that using static analysis to guide the load-/store instrumentation can lower the runtime overhead to the point that our approach is feasible for resilience techniques.

0.1 OPR: Deterministic Partial R&R

OPR involves the following steps (see Figure 1).

Record at full concurrency. The user first specifies the replay set, R_Set, a subset of threads that need to be replayed. A modified compiler is used to build a binary with recording instrumentation, tracking both `load/store` instructions, as well as communication operations (e.g. `Put/Get`). The instrumented binary is then executed at full scale on a modified UPC runtime system that records the execution. For any tasks within R_Set, we track

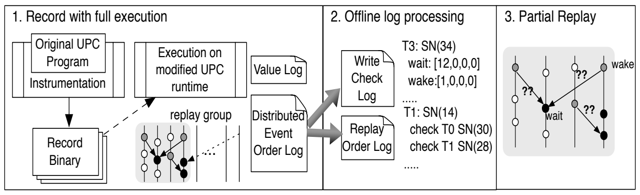

Figure 1: Overview of OPR.

`loads/stores` instructions into a value log, which contains the inputs for loads at different points. For any task within R_Set, we track `Put/Get` operations to tasks within R_Set into an distributed event order log. The event order log indicates an approximation of orders of conflicting operations accessing the global memory.

The behavior of any tasks outside R_Set, or the communication between R_Set and the outside world is not tracked.

In Figure 1, the shaded region indicates the replay group. In each thread, the white dots indicate read accesses that do not have value log entries; the black dots indicate read accesses that generate value log entries; the grey dots indicate write accesses. The arrows indicate detected event orders. We can see that some orders exist between write and read accesses, but the reads may not consume the values produced by writes, such relationship needs to be checked in replay phase. Also, some read accesses could get values produced by threads outside R_Set, such as the second black dot in the last thread in R_Set.

Log processing. The value log and order log are processed to enforce the replay order. Based on the distributed event order log, this pass generates a replay order log for each thread in R_Set. The event orders are translated into wait and wake vector clocks for the relevant operations so that threads in R_Set could collaboratively enforce the order present in the original execution. In addition, a write check log is generated for each thread so that it could try to match its own written values with remote read values in certain ranges at correct points in replay phase. We use this value based approach to infer communications between threads in R_Set because there is no explicit matching between senders and receivers in one-sided communication.

Replay only R_Set OPR only executes the threads in R_Set in the partial replay phase. The side effects of any other tasks can be reconstructed from the logs. Each thread reproduces the same execution by injecting the values in its value log at correct points. The operations from different threads are scheduled to execute in an order according to the replay order log. In addition, after a thread performs certain writes, it needs to check whether all the local writes so far could contribute to some read value log entries of remote threads. On a value match, a communication is assumed to happen between the two threads. This process is driven by the write check log. For each read log entry of a thread in R_Set, OPR could infer one of two possibilities: (a) the value is produced by a thread inside R_Set, if so, the specific thread is given; (b) the value is not produced by any thread inside R_Set. In Figure 1, the question marks indicate the value matching operation.

Now let us consider how does OPR work for the UTS example in (Listing **??**). Assume R_Set is $\{T_0, T_2\}$ and in a period of execution, T_0 steals from T_2 and T_3. In the record phase, in both steals, OPR will log the values of `s->stolen_work_addr` and `s->stolen_work_amt` at the correct time. In the replay phase, these values will be fed into T_0 at the same execution points. This ensures that T_0 is replayed correctly in isolation. In addition, based on the logs generated by the offline processing step the write operations in T_2 are executed before the read operations in T_0 that caused the exit of the while-loop. Furthermore, after writes in T_2 are performed, T_2 will check whether its writes performed so far could match a read value log in T_0. In our case, since T_0 indeed steals work from T_2, there will be matches for both values of `s->stolen_work_addr` and `s->stolen_work_amt`. Based on the matched values, OPR infers that the communication happened from T_2 to T_0.

In OPR, we use the principle of data-replay to ensure the correct replay of each thread in R_Set based on value log. We use order-replay and value matching to infer the communications between threads in R_Set. This design principle is critical since purely relying on order-replay requires replaying all threads (not satisfying requirement of partial replay). More importantly, due to non-atomic instrumentation, it is very challenging to generate precise event orders. The current approach could tolerate such imprecision because replay correctness does not depend on the event order. The imprecise event order only leads to false positives or negatives in communication inference but does not affect replay correctness.

References

[1] *Berkeley UPC. http://upc.lbl.gov.*

[2] *The NAS Parallel Benchmarks. Available at http://www.nas.nasa.gov/Software/NPB.*

[3] *UPC Home Page. http://upc-lang.org.*

[4] *MPI: A Message-Passing Interface Standard. Version 3.0.* Message Passing Interface Forum, 2012.

[5] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98, 1998.

[6] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[7] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly. In *Proceedings of the 26th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, November 2014.

[8] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987.

[9] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical Work Stealing on Manycore Clusters. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS)*, Oct 2011.

[10] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic Logging of Operating System Effects to Guide Application-level Architecture Simulation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, 2006.

[11] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, 2007.

[12] 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*, CGO '10, 2010.

[13] J. Sloan, R. Kumar, and G. Bronevetsky. Large Scale Debugging of Parallel Tasks with AutomaDeD. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[14] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, and G. Voelker. MPIWiz: Subgroup Reproducible Replay of MPI Applications. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 251–260. ACM, February 2009.