PaRV: Parallelizing Runtime Detection and Prevention of Concurrency Errors

Ismail Kuru[1], Hassan Salehe Matar[1], Adrián Cristal[2], Gokcen Kestor[2], and Osman Unsal[2]

[1] Koç University, İstanbul, Turkey
{ikuru,hmatar}@ku.edu.tr
[2] Barcelona Supercomputing Center, Barcelona, Spain
{adrian.cristal,gokcen.kestor,osman.unsal}@bsc.es

Abstract. We present the PaRV tool for runtime detection of and recovery from data races in multi-threaded C and C++ programs. PaRV uses transactional memory technology for parallelizing runtime verification and for buffering write accesses during race checking. Application threads are slowed down only due to instrumentation, but not due to the computation performed by runtime verification algorithms since the latter are run concurrently on different threads. Buffering writes allows us to recover from races and to safeguard against later ones.

1 Introduction

We present PaRV, a tool for runtime detection of and recovery from data races in multi-threaded C and C++ programs. We use components from transactional memory (TM) implementations in order to parallelize runtime verification and to buffer write accesses until they are determined to be free of races.

Concurrently with each application thread, a sibling thread in the style of [5] performs race detection using the Fasttrack algorithm[4]. This approach to parallelized runtime verification minimizes application slowdown. The application thread only experiences slowdown due to instrumentation. Once the sibling thread determines that the accesses within a block are free of races, the accesses in the buffer are committed to memory. If a race is detected, the block is rolled back, and extra synchronization is performed on variables experiencing races, which allows the execution to continue without race conditions. In its current form, our approach allows race-free execution of application binaries at a modest overhead even for legacy applications. With the availability of TM hardware in upcoming microprocessors and with a large number of cores expected to be available on processor chips, we expect our approach to have further reduced performance overhead and wide applicability for legacy applications.

2 Transactional Memory and Runtime Verification

Runtime verification slows down applications. For instance, race detection slows down C/C++ programs by 100 times or more. High overheads make post-deployment use of such runtime monitoring techniques infeasible. Even during

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 42–47, 2013.

pre-deployment testing and runtime verification, such high overheads make it unlikely that runtime verification techniques will be used continuously during all runs.

Transactional memory implementations contain highly optimized mechanisms for logging and buffering events, and, in the case of parallelized implementations of transactions, for efficient inter-thread communication between threads working on the same transaction. Hardware vendors have started providing hardware support for transactional memory, which will make approaches using TM more efficient in the near future.

A related approach is that of log-based architectures (e.g., [3], [6]) which provide on-chip hardware resources for reducing the runtime overhead that comes from monitoring executions. Differently from log-based approaches, our tool does not need any additional hardware support but can benefit from it. Differently from how [6] makes use of hardware TM, we do not make use of conflict detection and version number management – parts of a TM implementation that incur significant computational overhead. We use the high-performance FastTrack algorithm instead.

One important way our approach will benefit from hardware support for TM announced or provided by processor vendors is by obviating the need for software-based synchronization to protect race checking metadata, such as vector clocks. With compiler support available for TM hardware, our approach will immediately enjoy the benefit of improved performance due to TM hardware.

STM^2 [5] is a novel, multi-threaded STM design, where each application thread has a dedicated auxiliary ("sibling") thread performing STM operations such as validation of read-sets, bookkeeping and conflict detection. The communication between application and auxiliary thread is provided by a communication channel and atomic status variables. The communication channel is implemented with a single-producer/single-consumer, circular, lock-free queue where the application thread (producer) posts read and write messages that the auxiliary thread (consumer) retrieves and processes them. We use STM^2's queue to communicate read, write and synchronization operations from the application thread to the sibling thread carrying out race detection. We also use STM^2's write buffering and transaction commit mechanisms to delay writing to memory of writes until they are shown to be bug-free. Specifics of these are explained in the next section.

3 Tool Architecture and Implementation

Figure 1 shows position of PaRV relative to DynamoRIO-Dynamic instrumentation tool. The high level organization of the tool is as follows. Instructions performed by each application thread are instrumented using the dynamic DynamoRIO binary instrumentation framework [2]. Between every application thread and its corresponding sibling thread, there is a FIFO queue (figure 2) in the style of the STM^2 circular buffer that the application thread writes to and the sibling thread reads from. On the application thread, read and write accesses and synchronization operations are instrumented such that for each of

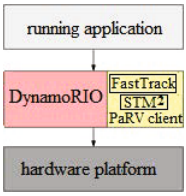

Fig. 1. Architecture of PaRV

these instructions executed, an *event* is placed on the FIFO queue. The sibling thread removes events from the queue and is able to carry out race detection for the sequence of instructions carried out by the application thread in this way.

The sequence of instructions performed by each thread are divided into non-overlapping portions called *consistency block*s using DynamoRIO binary instrumentation. Every synchronization event is in a consistency block by itself. The sequence of instructions performed by an application thread between two synchronization events constitute a block otherwise. The application thread and the sibling thread synchronize at consistency block boundaries. The application thread buffers all write accesses it performs. For consistency blocks that do not contain synchronization operations, when the sibling thread signals to the application thread that the processing of the block is complete, and detects no concurrency errors, the application thread commits the writes in the buffer to memory. For consistency blocks consisting of synchronization operations, the application waits for the sibling thread to complete processing the consistency block before it actually performs the synchronization operation. This is necessary for the runtime verification carried out by the sibling threads to have the same happens before relation as the execution produced by the application threads. Before using DynamoRIO to realize the implementation we tried our approach with PIN. However,with PIN we could not do some of the approaches discussed.

3.1 Runtime Instrumentation with DynamoRIO

Using DynamoRIO, the write and read accesses and synchronization operations performed by application threads are modified.

A write access (store instruction `ins`) is instrumented to implement the following steps. First, the address and the value to be written are extracted from `ins`. Then, an entry is written into the write buffer of the application thread, and an event corresponding to the write access is placed on the FIFO queue. The write instruction is then skipped. This is necessary, since we only want to commit to main memory writes determined to be free of concurrency errors. The write buffer implementation is borrowed from STM^2

```
addr = get_destination ( ins );    val = get_value ( ins );
write_to_buffer( addr, val);    enqueue_write_event( addr);
skip_instruction ( ins );
```

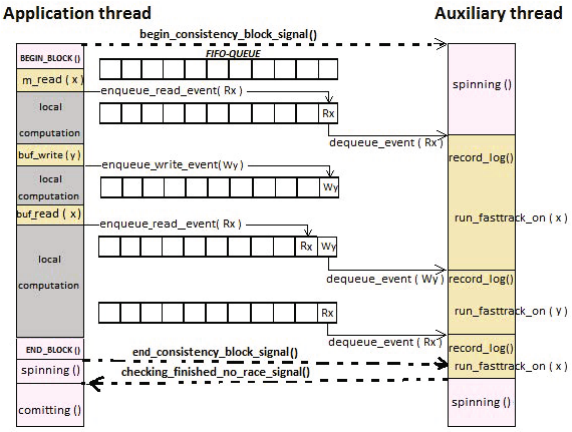

Fig. 2. Application-auxiliary thread interaction

A read access (load instruction `ins`) is instrumented so that a variable that was written to earlier by the current consistency block gets its value from the write buffer. Other reads get their value from the main memory.

```
read_from_local_write_buffer_or_mem ( ins );
addr = get_memory_operand ( ins );
enqueue_read_event ( addr );
```

When a consistency block ends, the application thread waits for the sibling thread to set an atomic signal to indicate completion of runtime verification for the curent consistency block. At that time, the write buffer is committed to main memory. Unlike the commit phase of an STM implementation, we do not need to acquire locks for the variables in the write buffer, since we are not carrying out conflict detection between consistency blocks in the sense of TMs. If no race has been detected by the sibling thread, then write buffers can safely be written to memory, since there are no concurrent racy writes. This is a significant factor in reducing the instrumentation overhead below what an STM would experience.

Before every synchronization operation, we end the ongoing consistency block if there is one, and then start a new one. We put on the FIFO queue an event representing the synchronization operation. When the sibling thread is done processing this event and notifies the application thread by setting an atomic variable, the application thread continues, performs the synchronization operation, and starts the new consistency block.

3.2 Detecting and Recovering from Races

Each sibling thread applies to the stream of events it receives from the event FIFO queue the FastTrack race detection algorithm [4]. FastTrack is an efficient, precise race detection algorithm. The algorithm is described by providing the updates and checks performed by each thread for each memory access or synchronization operation. In our tool, differently from the original FastTrack,

the application thread only records the events in the FIFO queue. The race detection computation is performed on the sibling thread for each event as it is removed from the FIFO queue. We implemented FastTrack in C based on the original implementation. The shared variables (e.g. vector clocks and epochs) used by FastTrack are protected by mutual exclusion locks. The sibling thread notifies the application thread of races or race-free completion of consistency blocks by setting atomic variables.

By buffering write accesses until the end of a consistency block, we are able to prevent racy writes from being written to memory, and racy reads from affecting later code. At the end of a consistency block, if the sibling thread signals a detected race condition, the consistency block is aborted (the write buffer discarded) and retried. The sibling thread notifies the application thread of the set of variables that experienced a race condition during the last execution of the consistency block. The application thread, when retrying the block, wraps each access to a racy variable x by an acquire and release of the lock that protects VC_x, the vector clock of x. Since the last access by another consistency block to x was followed by the sibling thread's access to VC_x, this ensures a happens-before relationship between the accesses and prevents a race condition. After a race is detected on x, all later accesses to x by application threads are protected by VC_x. By doing this for only variables that experience a race, we keep the performance overhead of our approach low.

4 Related Work

PaRV builds on research in the areas of transactional memory and dynamic race detection. It also bears similarities to approaches in the architecture literature for instrumenting and logging program executions, parallelizing dynamic monitoring, containing and recovering from errors encountered. In the following, we contrast PaRV with these approaches.

ParaLog [8] extends work on log-based architectures [3] provide hardware support for instrumenting, logging and monitoring executions of multithreaded programs. Techniques in ParaLog not only reduce the application slowdown due to instrumentation and logging, but also allow, similarly to PaRV, parallelized monitoring algorithms to be run on separate resources from the application, thus further reducing slowdown. ParaLog involves significant changes to processor and memory architecture. It accomplishes efficient tracking of ordering of events from different threads by monitoring cache coherence traffic. PaRV works on currently available, stock microprocessors, but If a platform provides LBA support, PaRV would incur much less slowdown as well.

Race-detection depends critically on, and almost entirely consists of tracking inter-thread dependencies precisely, and the multiple threads in the monitor accessing the per-address and per-thread metadata atomically. The hardware support in ParaLog directly targets efficient implementations of these operations. Taking an alternative approach, PaRV aims to reduce race-detection slowdown as much as possible in the absence of hardware support for monitoring. Differently

from ParaLog, PaRV uses TM technology to prevent races, and explicitly inserts extra synchronization into the program for avoiding later races.

The authors in [1] present the KUDA tool, which, similarly to PaRV, separates race detection from application execution threads using kernel threads in the GPU as helper threads. Differently from KUDA, PaRV synchronizes the application and helper threads so that race detection does not lag behind. This is essential for prevention of and recovery from races, two more features that distinguish PaRV from KUDA. KUDA also parallelizes race detection further than one helper thread per application thread in order to make use of the high degree of parallelism provided by the hundreds of cores on a GPU.

Veeraraghavan, et al in [7] present the Frost tool that addresses detection and prevention of data races by running multiple replicas of an application using complementary schedules. Races are detected by comparing states reached by different replicas, instead of processing event sequences. While providing significant reduction in slowdown, this approach suffers from two key weaknesses. First, for an application with faulty synchronization, it is quite possible that no schedule leads to race free execution. PaRV addresses this problem by adding synchronization to the program as needed. Second, race detection in Frost is imprecise. PaRV uses the FastTrack algorithm for precise detection of races.

References

1. Bekar, U.C., Elmas, T., Okur, S., Tasiran, S.: Kuda: Gpu accelerated split race checker. In: Workshop on Determinism and Correctness in Parallel Programming (WoDet), London, England, UK (March 2012)
2. Bruening, D.L.: Efficient, transparent and comprehensive runtime code manipulation. Technical report (2004)
3. Chen, S., Falsafi, B., Gibbons, P.B., Kozuch, M., Mowry, T.C., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G.R., Lin, B., Schlosser, S.W.: Log-based architectures for general-purpose monitoring of deployed code. In: Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID 2006, pp. 63–65. ACM, New York (2006)
4. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. SIGPLAN Not. 44, 121–133 (2009)
5. Kestor, G., Gioiosa, R., Harris, T., Unsal, O.S., Cristal, A., Hur, I., Valero, M.: Stm2: A parallel stm for high performance simultaneous multithreading systems. In: 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 221–231 (October 2011)
6. Sánchez, D., Aragón, J.L., García, J.M.: A log-based redundant architecture for reliable parallel computation. In: HiPC, pp. 1–10. IEEE (2010)
7. Veeraraghavan, K., Chen, P.M., Flinn, J., Narayanasamy, S.: Detecting and surviving data races using complementary schedules. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 2011, pp. 369–384. ACM, New York (2011)
8. Vlachos, E., Goodstein, M.L., Kozuch, M.A., Chen, S., Falsafi, B., Gibbons, P.B., Mowry, T.C.: Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In: Proceedings of the Fifteenth Edition of ASPLOS, ASPLOS 2010, pp. 271–284. ACM, New York (2010)