# Race Detection for Stable Multithreading System

Xinan Xu, Heming Cui, Junfeng Yang

Department of Computer Science, Columbia University

**Abstract**

One of the most noticable problems in multithreaded programs today is data race. They are hard to diagnose and fix due to the nondeterministic nature of the current operating system. Much effort has been made in the past, proposing various tools to conquer this problem, however, usually suffer from huge performance penalty. Here, we propose and implement a simple, practicle race detection algorithm on Stable Multithreading system which not only makes data race detection process deterministic and reproducable, but also largely reduce the performance overhead.

**Key words.** Thread Sanitizer, Data Race Detection, Stable Multithreading

# 1 Introduction

Data race has always been a serious problem in multithreading software as it is not only hard to detect, but also hard to diagnose and reproduce. The main reason for this is the nondeterministic of the multithreading system scheduler, which arbitrarily assign different thread interleavings for different executions. Such scheduler would obstruct the traditional race detection tool to stably report bugs as some bugs may appear in a rarely executed schedule which is not discovered during stress tests. Even when a rare bug is discovered, Record and Replay technique is needed to reproduce such bug. Moreover, traditional data race detection always incurs extreme performance penalties as it not only needs to intercept each memory access, but also trace the pthread synchronization records for a later happens-before relation evaluation.

Substituting the system scheduler with a Stable Multithreading scheduler would enforce a total-order for synchronizations, which shrinks the number of schedule from extraordinary number to one. This not only improves the stability of data race detection process, makes the bug easier to reproduce without Record and Replay, but also simplifies the data race detection algorithm: Since the scheduler for the Stable Multitherading system is predefined, the happens-before relation is also well defined by a global turn number. Instead of recording pthread synrhonization events and/or mutex sets, only the turn range is needed for evaluating the happens-before relation. In this report, we implemented our data race detection tool based on Thread Sanitizer 2, a race detection tool included in the llvm development branch. We applied our tool on PARROT, a practicle and fast Stable Multithreading system.

# 2 Algorithm

## 2.1 Definition of Epoch

The order number of the turn, or "epoch" is defined globally, instead of thread-local manner of other dynamic race detection tool. Based on the model of Parrot, which maintains a runnable queue and a waiting queue, the global epoch is increased after the thread has get the turn. This means some synchronization operation for a thread will getTurn twice. For example, when a thread fails to acquire a mutex lock, it will get the turn when it tries the first time to get the lock, and it will get the turn for the second time when it is waken from the waiting queue. In such cases, the global epoch will be increased twice.

```
#define SCHED_TIMER_START \
  record_rdtsc_op("GET_TURN", "START", 2, NULL); \
  _S::getTurn(); \
  record_rdtsc_op("GET_TURN", "END", 2, NULL); \
  if (pthread_self() != idle_th) \
    stat.globalTurnNum++;
...
#define SCHED_TURN_INC \
  if (pthread_self() != idle_th) \
    stat.globalTurnNum++;
```

Listing 1: Additional codes in PARROT runtime

## 2.2 Predicting the Epoch Range

The best thing of the stable multi-threading system is that, every memory access is located within an epoch range, and such range could be pre-calculated at the previous synchronization point. Here, each thread is maintaining an epoch range, [epoch_begin, epoch_end]. The epoch_begin is simply the epoch of the latest synchronization operation for the current thread, and epoch_end is the sum of the epoch_begin and the number of the runable threads. The idle thread should be exclueded:

```
epoch_end = epoch_begin + runQueue.size() - idle_thread.exists();
```

This formular is well suited for PARROT system, and is proved and tested to be correct over all pthread syncrhonization operations. Consider the simple round-robin scheduler provided by PARROT for its runnable threads, each one of the thread has to execute the same of synchronization as the number of runnable threads, as the token has to been passed one round back to the original owner. More complex situations are categoried as following:

1. A thread is removed from the runnable queue, either put into waiting queue or terminated. Since this thread will be waiting, it will not have any further memory access,

and the calculation of the epoch_end is unnecessary(and also meaningless). The number of the runnable queue is decreased by 1, so it will not effect the correctness of other threads.

2. A thread is waken from the waiting queue, and put back to the running queue. The number of runnable threads is increased by 1, however, this does not affect the waiting time for other threads, as they are waiting before this thread.

3. A new thread is created and put at the back of the running queue. This is a same situation as the above.

## 2.3 Race Detection

### 2.3.1 Overview

For each memory access, it is within an deterministic epoch range. The shadow state of the current memory address maintains the epoch range for both logically latest read and write. (Here, logically latest means the trunk with a highest epoch range begin, which is different from physically latest), The instrumentation of the memory access will involve the comparison for the epoch ranges. If the current memory access is write, its epoch range will be compared with the logically later one of the stored read and write epoch ranges. If the current memory access is read, it will be compared with the write epoch range. If the comparison gives an intersect, the race report will be generated.

### 2.3.2 Data Structure

Similar as thread sanitizer, each byte in memory corresponds to a shadow state which contains additional information about the previous memory accesses for that byte.

```
struct shadow_state{
  range<int,int> read_epoch;
  range<int,int> write_epoch;
};
```

Listing 2: Structure of a shadow state

In each shadow memory state, two epoch ranges are stored for both previous read and write memory access.

### 2.3.3 Core Algorithm

After restricting each memory access within an epoch range. Data races are considered potential if two memory accesses have the same memory address and happens in epoch ranges which intersect with each other, but are not the same (thus not from the same thread). Here, the intersect function usually involves one comparison as the epoch_begin of the former memory access must be less than the epoch_end of the current memory access. The update_write function involves one memory store operation, which uses the current epoch

range to overwrite the shadow state. The update_read function is a little more complicated. It does a union operation for both ranges which are unionable, and a simple overwrite for ranges that are not unionable. It involves up to two memory stores and one comparison. As a result, the shadow state is always storing a logically latest epoch range for write memory access, and a logically latest continuous epoch range for read access. Total operations inserted for one memory access is 2 memory loads, up to 5 comparisons and 2 memory stores.

```
void memory_access(addr, rw, pc) {
  shadow_state shadow = get_shadow_state(addr);
  range<int,int> cur_epoch = get_current_epoch();
  switch (rw) {
    case read:
      if(cur_epoch == shadow.read_epoch)
        return;
      if(intersect(cur_epoch, shadow.write_epoch)
        reportRace();
      if(cur_epoch > shadow.read_epoch)
        shadow.update_read(cur_epoch);
      break;
    case write:
      if(cur_epoch == shadow.write_epoch)
        return;
      if(intersect(cur_epoch, shadow.read_epoch) \
        || intersect(cur_epoch, shadow.write_epoch))
        reportRace();
      if(cur_epoch > shadow.write_epoch)
        shadow.update_write(cur_epoch);
      break;
  }
}
```

Listing 3: Pseudo code for race detection algorithm

Storing the logically latest memory access information only is very aggressive, however, is correct. As we don't want to miss the bug by only maintaining the latest memory access information, we would like to prove that, if a current memory access races with any previous memory accesses, it must also race with the logically latest one stored in the shadow state.

We use the negative approach to prove this: Consider a memory access that does not race with the logically latest memory access, two ranges either must not intersect with each other, or they have the same value indicating they are from the same thread. In the first case, non-intersection means the former memory access strictly happens-before the current memory access, and since it is also the logically latest one, any previous memory accesses should also strictly happens before the current memory access. In the second case, if the former memory access has the same epoch range with the current access, then the race between the current access and all previous access has already been reported when the former memory

access happens and is write access. For read access, we are maintaining the logically latest continuous epoch range to avoid the second case happening. [1]

## 2.4 Complexity

The number of instructions instrumented for each memory access is constant. The total runtime complexity is $O(m)$, m denotes the number of total memory accesses. The shadow memory overhead is 8x, as each byte is mapped into 8-byte shadow state. For Thread Sanitizer 2, the runtime complexity is $O(m)$, however, it saves 4 memory access histories for each memory address and has possibility to miss bugs. The runtime complexity for Thread Sanitizer 1 is $O(m * n^2)$, n denotes the number of threads. For each memory access, Thread Sanitizer 1 will update the segment set for that memory address and determine whether two segment sets make a race.

## 2.5 Misc.

### 2.5.1 Bug report

To present a useful bug report, we use several modular-based hash tables to save the information for racing memory accesses. As same as the original version, those hash tables does not resolve conflict problem, and the previous information will lost during conflicts, which result in rarely incomplete bug report.

### 2.5.2 Atomic load and store

To ensure there is no data race within our algorithm, the load and store of the shadow state is atomic, otherwise, nondeterministic bug report was observed.

### 2.5.3 Idle thread

PARROT is creating background and idle thread for its own purposes. Detecting data race within those threads are not necessary and is avoided. To ensure the thread ID matching of PARROT runtime and Thread Sanitizer runtime, a conditional offset has been used. A better approach is to create a thread ID map between two systems.

# 3 Evaluation

To evaluate how much our algorithm cutdown the performance overhead, we compared it with the Thread Sanitizer 2 shipped with LLVM 3.3. Thread Sanitizer is a dynamic race

[1] This is to avoid a case: The first access is read; The second access is also read and intersects with the first one; The third access is write and happens in the same epoch range as the second one. If we use the same mechanism as write, the second access will overwrite the first access and hide the race between the first and third access. This was observed in Radix

detection tool uses a hybrid algorithm combining the happens-before algorithm and lockset algorithm. The complexity for each memory access instrumentation is $O(1)$, which is the same as ours. However, as it is saving four memory access histories in one shadow cell, it needs to detect the data race looping all those histories. In our experiments, we disabled the bug reporting to reflect the overhead of the detection part. We also used the small input defined by all benchmark programs. Those experiments on pbzip2, splash2 benchmark suite and phoenix benchmark suite shows 42.95% overhead cutdown in average.

| Program | TSAN2 overhead | TSAN2+SMT overhead | Overhead Cutdown (%) |
|---|---|---|---|
| pbzip2 | 45.96 | 35.02 | 23.80 |
| pbzip2 -d | 26.53 | 17.29 | 34.83 |
| fft | 6.58 | 3.05 | 53.65 |
| barnes | 24.32 | 22.68 | 6.76 |
| lu_cb | 30.54 | 12.86 | 57.88 |
| lu_ncb | 29.51 | 13.51 | 54.23 |
| ocean_cp | 21.11 | 10.18 | 51.80 |
| ocean_ncp | 14.06 | 9.07 | 35.48 |
| radix | 6.75 | 4.28 | 36.54 |
| raytrace | 63.43 | 41.86 | 34.00 |
| histogram | 99.92 | 46.97 | 52.99 |
| linear_regression | 7.01 | 2.69 | 61.66 |
| matrix_multiply | 10.46 | 4.01 | 61.62 |
| pca | 25.98 | 10.05 | 61.31 |
| string_match | 30.08 | 25.02 | 16.85 |
| word_count | 5.37 | 3.02 | 43.77 |
| Average | | | 42.95 |

Table 3.1: Evaluation of performance overhead cutdown

# 4 Future works and current issues

## 4.1 Atomic instruction

Atomic instruction is not yet supported, as it would double the memory usage for our algorithm. Using bit flag for marking atomic instruction will make our race detection unsound. As atomic instructions are not prevailing currently, we choose to ignore atomic instruction right now.

## 4.2 Bug reports

Since the incompatibility of our shadow state structure with the original LLVM bug reporting design, we provides the bug report in a simpler manner. Instead of providing the backtrace of racing memory accesses, we only provide the toppest instruction and source code line for

racing memory accesses. It will not be hard to implement a similar tracing system for race detection.

## 4.3 Epoch overflow

In the current implementation, to save a shadow state within a 64bit interger, we use 16bit interger for the epoch number, which limits the maximum number of epoch to be 65535. It is possible to overflow the epoch number if the program is synchronization intensive. Such issue has been observed in raytrace. A fix to this problem is to either increase the size of a shadow state, or to replace the epoch_end with epoch_range, which needs less bits and gives more bits to the epoch_begin. This method is used before git commit c946cd0, however, presents a slightly worse performance.

## 4.4 Other concurrency bug detection

The idea of this algorithm could be easily applied to other types of concurrency bug detection such as order violation. Additional bits need to be added into the shadow state indicating whether the memory has been allocated/freed. The intersection or a flipped order of the epoch range is considered as a order violation.