ATLANTIS PRESS

Advances in Computer Science Research, volume 63

International Conference on Artificial Intelligence and Engineering Applications (AIEA 2016)

A Novel Data Race Detection Approach based on Buddy Memory Allocator

Zhengyang Liu [a], Hua Zhang [b,*]

State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, 100876, China

[a]y@bupt.edu.cn, [b]zhanghua_288@bupt.edu.cn

Corresponding author: Hua Zhang

Keywords: Data race, Memory model, Dynamic analysis, Computer security.

Abstract. As a common problem of multi-core parallel programs, the problem of data race has been paid more and more attention in recent years. In this paper, a dynamic detection approach for data race problem detection is proposed. By introducing a new metadata storage based on the buddy memory allocator, the metadata access performance is improved significantly. A specific implementation of the approach based on LLVM compiler infrastructure is made. The experimental results show that the proposed approach can reduce the time cost of dynamic race detection and achieve 2x-5x performance on the Olden benchmark.

1. Introduction

Since the invention of multi-core processors, personal computers, laptops, mobile phones and other terminals have the ability of multi-core processing. The extensive application of multithreaded programs brings data race problems, which is a situation that two or more threads simultaneously access the same memory object, with at least one write operation. Data race detection is divided into two kinds, the static analysis approach and the dynamic analysis approach. Due to the undecidability of static analysis, accurate static analysis approach of data race problems is proved to be an NP-hard problem [1], and practical static analysis is usually an incomplete approximation algorithm. Common dynamic analysis approaches of data race detection have the overhead of 10x-30x, their high cost seriously compromise the practicality.

In order to reduce the cost of dynamic data race detection and balance the efficiency and scale, this paper proposes an efficient data race dynamic detection scheme. By storing the metadata of the corresponding memory object in the tail of the memory object, the detection algorithm can quickly detect whether the memory access exists data race problem. We implemented the new method using LLVM compiler infrastructure [13] and the SAFECode compiler [9].

The rest of this paper is organized as follows. Section 2 provides background on buddy memory allocator. Section 2 presents the design of our approach, and Section 3 describes our implementation within the LLVM compiler. Section 4 describes our performance evaluation of our approach, Section 6 discusses related work, and 7 concludes with a discussion on future work.

2. Metadata model

Metadata and memory objects are one-to-one correspondence. The authors of [5] [6] [12] note that the metadata access overhead is the main overhead in race detection algorithms. Jones et al [4] use the linked list as the data structure of metadata, with retrieving the complexity O (n) theoretically, however, the time overhead is only 80x. Dhurjati et al [5] [6] use the splay tree as the data structure of metadata, with the retrieving complexity O (log2n), the actual time overhead of 20x. Thread Sanitizer [10] [12] maps the whole memory space, and a continuous block of memory is reserved as a linear table, with the O (1) access complexity. However, this method does not use the data locality optimization, the practical time cost is 8x-15x. In order to improve the retrieval performance of metadata, we design a new metadata memory model based on buddy memory allocator. Our memory model takes full advantage of the computer system for the spatial consistency of data optimization, experimental results show that the new memory model will reduce the time cost to only 2x-5x.

2.1 Memory Model

In order to improve the performance of data race detection, we use the buddy memory allocator, all memory objects aligned to the power of 2. For example, for x = malloc (200), we will transform the program to allocate a 256-byte memory space. We follow the same assignment for global variables and stack space. Because of the alignment and padding, we can use only one byte to store the allocation space of a memory object.

e = log2 (Size) (1)

In this paper, we use an contiguous array to store the encoded size of the allocated space, because each record occupies only one byte of space, the cost of the adjacency array is 1/ SlotSize. For a given address p, we can query on the contiguous array to obtain the binary logarithm of the memory object length.

e = SizeTable [p>> SlotSize] (2)

We need to store the corresponding metadata to achieve dynamic data race analysis. Due to the ubiquity of data locality, common computer systems provide optimizations for nearby data access. The EventList will be stored in the memory cell after the fill area, with the memory space layout shown in Fig. 1.

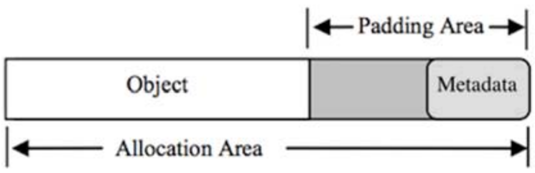

Fig. 1 Memory layout

After retrieving e, we can quickly determine the starting address of the memory object and the starting address of the metadata as well as the allocation size.

Size = 1 <<e (3)

base = p& ~ (size - 1) (4)

MetaData = *(base + (1 <<e) – sizeof(MetaData) (5)

2.2 Metadata design for data race detection

We define the following data structures to represent events for multithreaded memory accesses.

Tid: A unique identifier for a thread in the running program.

AccessType: A data structure that records whether or not an access is a write access.

TimeStamp: A timestamp identifying the access time.

Event: It records the data access event of a memory structure, which is a triple with form {Tid, AccessType, TimeStamp}.

EventList: [Event1, Event2, ...] records the data structure of a recent memory access array event for a memory object.

IsLocked: It records whether a memory object is locked or not.

MetaDataObject: <EventList, IsLocked> records the metadata of a memory object.

Object: For each memory object, a corresponding metadata is attached to record the multi-threaded memory access trace. It is a triple with form <MemoryObject, MetaDataObject>.

3. Race detection algorithm

3.1 Access sanity check

The access sanity check algorithm is used to check the legitimacy of an access event and to record the information in the metadata area for subsequent checks. The access checking algorithm accepts the Tid, TimeStamp, and AccessType of the current access, and compares the access information with the metadata of the memory unit. If there is data race, the program is interrupted and the user is alarmed. If the data race does not exist then the follow-up procedures normally.

Algorithm 3 is the pseudocode for address sanity check.

```
Algorithm 3. Access sanity check algorithm
Algorithm sanity_check (*p, Tid, Timestamp, AccessType)
    size = 1 << table [p >> SlotSize];
    base = p & ~ (size – 1);
    last_access = rc_table [base >> SlotSize];
    epoch = last_access. epoch;
    thread_id = last_access. thread_id;
    isWrite = last_access. isWrite;
    If current. epoch >> epoch||thread-id == current. thread-id {
        rc_table[base>>SlotSize] = current;
        return;
    }
    If (current. isWrite == false && isWrite == false) {
        rc_table[base>>SlotSize] = current;
        return;
    }
    EmitWarning ();
```

3.2 Memory Access

For a memory access event, we need to determine the legitimacy of its access, that is, whether it will cause data race, and stores the visit information in the corresponding metadata area.

We perform algorithm 4 for any read operation.

```
Algorithm 4. read operation
Algorithm load (*p, Tid, Timestamp, READ):
    RDPD_check (*p, Tid, Timestamp, READ)
    return load (*p)
```

We perform algorithm 5 for any write operation.

```
Algorithm 5. write operation
Algorithm Store (*p, value, Tid, Timestamp, WRITE):
    RDPD_check (*p, Tid, Timestamp, WRITE):
    return store (*p, value)
```

4. Implementation

In this section, we implemented our approach on the Linux 3.6 64-bit operating system and LLVM compiler framework [14]. We select 16 bytes as SlotSize. As multi-threaded data race exists only in the user space memory, in order to save costs, we only check user space memory accesses. For 64-bit Linux operating system, the user space memory address is from 0x0000000 00000000 to 0x00007FFF FFFFFFFF. Thus, the length of SizeTable is 1 << 43, calculated by Size >> log (SlotSize). We uses the mmap() system call provided in posix standard [2] to allocate the space for the SizeTable. The overall system architecture is shown in Fig. 2.

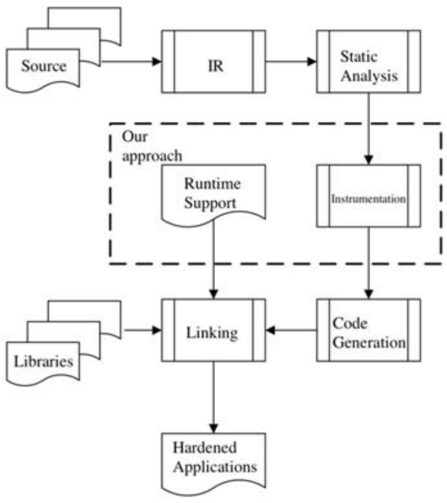

Fig. 2 System Architecture

5. Performance Evaluation

We used an Intel E5-2620v2, 16GB RAM machine as the experimental environment, which is running the Ubuntu 16.04 amd64 server operating system, we run the Olden benchmark [13], and compared our scheme with FastTrack, Fig. 3 is the benchmark result.

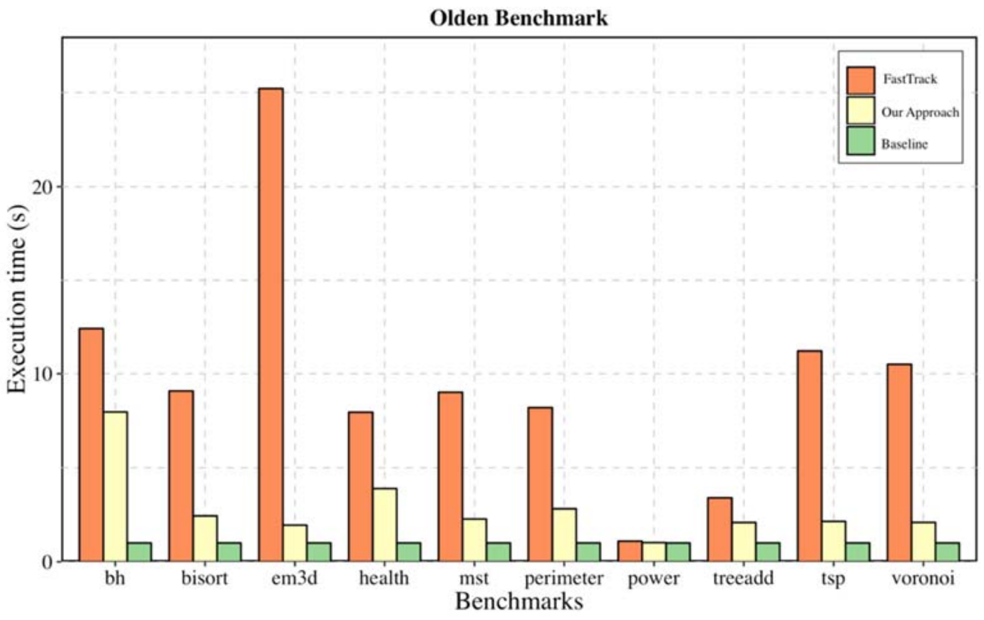

Fig. 3 Benchmark result

Fig.3 depicts that our approach has a 2x – 5x overhead, outperforms the FastTrack approach.

6. Summary

This paper studies the dynamic detection of multi-thread data race, and introduces a novel dynamic race detection scheme. The scheme improves the access performance of the memory object metadata by using the memory allocation method based on the buddy memory allocator. The experimental results show that the proposed scheme is practical and effective. The future work is to further improve the efficiency of the algorithm, extend it to more general multi-thread data dynamic detection, and combine the race detection with other memory access problem detection.

Acknowledgments

This work is supported by NSFC (Grant Nos. 61300181, 61502044), the Fundamental Research Funds for the Central Universities (Grant Nos. 2015RC23).

References

[1] William Landi. Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS) 1.4 (1992), p. 323-337.
[2] David R Butenhof. Programming with POSIX threads. Addison-Wesley Professional, 1997, p. 103-105.
[3] Laszlo Szekeres, Mathias Payer, Tao Wie, et al. Sok: Eternal war in memory. IEEE Symposium on Security and Privacy (SP). San Fransisco, CA, USA, 2013, p. 45-53.
[4] Cormac Flanagan, Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. ACM Sigplan Notices. Vol. 44 (2009) No. 6, p. 121-133.
[5] Dinakar Dhurjati, Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead.ACM International Conference on Software Engineering. Shanghai, China, 2006, p. 162-171.
[6] Richard WM Jones, Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. International Workshop on Automatic Debugging, Linköping, Sweden, 1997, p. 13-26.
[7] Periklis Akritidis,Manuel Costa, Miguel Castro, et al. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. USENIX Security Symposium. Montreal, Canada, 2009, p. 51-66.
[8] Baozeng Ding, Yeping He, Yanjun Wu, et al. Baggy bounds with accurate checking. IEEE 23rd International Symposium on Software Reliability Engineering Workshops. Dallas, TX, USA, 2012, p. 195-200.
[9] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. ACM SIGPLAN Notices. Vol. 41 (2006) No. 6, p. 320-331.
[10] Konstantin Serebryany, Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. Workshop on Binary Instrumentation and Applications. New York, NY, USA, 2009, p.62-71.
[11] Baris Kasikci, Cristian Zamfir, and George Candea. RaceMob: crowdsourced data race detection. ACM Symposium on Operating Systems Principles. Farmington, PA, USA, 2003, p. 406-42.
[12] Konstantin Serebryany,Derek Bruening, Alexander Potapenko, et al. AddressSanitizer: a fast address sanity checker. USENIX Annual Technical Conference. Boston, MA, USA, 2012, p. 309-318.
[13] Information on http://www.cs.princeton.edu/-mcc/olden. html.
[14] Chris Lattner, Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. IEEE International Symposium on Code Generation and Optimization. Palo Alto, CA, USA, 2004, p. 75-86.
[15] Serebryany, Konstantin, Alexander Potapenko, Timur Iskhodzhanov, et al. Dynamic race detection with LLVM compiler. International Conference on Runtime Verification. San Francisco, CA, USA, 2011, p. 110-114.