J Supercomput (2016) 72:1645–1659
DOI 10.1007/s11227-016-1691-1

CrossMark

DRDDR: a lightweight method to detect data races in Linux kernel

Yunyun Jiang[1] · Yi Yang[1] · Tian Xiao[1] · Tianwei Sheng[2] · Wenguang Chen[1]

Published online: 10 March 2016
© Springer Science+Business Media New York 2016

Abstract Data races in parallel programs are notoriously difficult to detect and resolve. Existing research has mostly focused on data race detection at the user level and significant progress has been made in this regard. However, a very few efforts have attempted to address OS kernel level races. In contrast to user-level applications, the code in kernels uses vastly more complicated synchronization mechanisms, including different kinds of locks, hardware and software interrupts, widely used signal/wait primitives, and various types of low-level shared resources. It is therefore difficult to apply detection methods designed for user-level applications to identify OS kernel level races. In this paper, we present a new detection tool that is able to effectively detect race conditions in the Linux kernel environment. We use a dynamic detection approach, employing hardware debug registers available on commodity processors, to catch races on the fly during runtime. Preliminary experimental results show that our tool can effectively identify real data race instances.

Keywords Data race detection · Linux kernel · Synchronization · Debug register · Sample

1 Introduction

Parallel programming stands to play a critical role in improving computing performance in the near future. Writing parallel programs, however, is complicated error-prone work. One of the most elusive concurrency bugs, notoriously hard to

✉ Wenguang Chen
cwg@tsinghua.edu.cn

[1] Department of Computer Science and Technology, Tsinghua University, Beijing, China

[2] Department of Computer Science and Engineering, University of California, San Diego, USA

detect and fix, is the data race condition which occurs when two threads (at least one of which is a write operation) simultaneously access the same shared memory space without proper synchronization constraints. Data races not only affect program results, leading to incorrect output, but may also induce system behaviors, such as halting or crashing the system. Examples include several serious incidents such as accidents of the Therac-25 radiation therapy device [1], the severe 2003 blackout in North America [2], and Nasdaq's Facebook glitch [3]. This pattern of events will persist due to widespread deployment of multi-threaded programs, with more and more accompanying, unforeseen consequences.

Considerable research [4–7] has been conducted on data race detection and significant progress has been made. However, most research efforts have focused on user-level data races and very little work has been done at the OS kernel level. Compared with in user-level applications, race conditions in the kernel may result in more severe consequences such as system crashes and massive data loss. This is because all applications stand to be affected if the OS itself is buggy or crashes.

Furthermore, in contrast to user-level applications, kernel-level code employs vastly more complicated synchronization mechanisms, including different types of locks, hardware and software interrupts, widely used signal/wait primitives and low-level shared resources. This makes it difficult to apply detection methods originally designed for user-level applications in this case.

DataCollider [8] is the first detection tool aimed at identifying data race conditions in the kernel. This solution uses the breakpoint functionality of debug registers to catch races on the fly. DataCollider bypasses the traditional detection mechanism by employing a ubiquitous structure debug register in modern hardware and checks the multi-threaded kernel codes for data races in an efficient manner. However, DataCollider is designed for the Microsoft Windows operating system, which is proprietary and cannot therefore be applied directly in further related research efforts. In this context, we present DRDDR (data race detection using debug register), a lightweight dynamic data race detector, the first work to apply the debug register-based race detection method to the Linux operating system. We furthermore discuss our practical experience with detecting race conditions in the Linux kernel and hope this information will prove of value to both the research and industry communities.

We implement DRDDR in the kernel module of a 64-bit Linux system based on x86 architecture. To verify the functionality of our tool, we construct a micro-benchmark and reproduce two confirmed data race bugs in a real-world system. Finally, we deploy DRDDR in the file system module of the kernel and identify certain new race conditions.

The rest of this paper is organized as follows: we provide an overview of the DRDDR framework in Sect. 2 followed by implementation details in Sect. 3. In Sect. 4 we describe our experiment and evaluate the performance of DRDDR. We discuss prior work in this domain and differentiate our research in Sect. 5. In conclusion, we discuss possible directions for future work in Sect. 6.

2 Overview of DRDDR

DRDDR samples a small amount of memory access operations as detection candidates and utilizes code breakpoints and watchpoints in modern computer architecture to detect race conditions. Our solution does not add extra runtime overhead to the non-sampled code regions, and, by choosing a low sample rate, is able to conduct the detection operation with negligible overhead.

The essential workings of DRDDR are depicted in Fig. 1. Our solution overcomes the aforementioned issues in kernel data race detection as follows: In a series of parallel programs, DRDDR first selects some memory access instructions at random and samples a small number of them to insert code breakpoints. The programs are then allowed to run normally. During the execution, when a code breakpoint is triggered, like *Thread 1* in the figure, DRDDR opens a short time window, sets a watchpoint on the sampled memory address using the debug register, and briefly monitors. If another thread accesses the critical shared memory data in this period, like *Thread 2* in the figure, the watchpoint is triggered and DRDDR reports a data race bug.

We introduce here a synthetic test case to further clarify the detection principle of the algorithm. Figure 2 depicts the test case *RWrace*, which has two threads that periodically and concurrently read/write the shared object `ptr`. *Thread 1* writes the variable and *Thread 2* read it, the value of the variable is rewritten by a random number in every loop operation. Since the whole procedure has no synchronization constraints whatsoever, any two read/write operations of the thread technically constitute a data race. *RWrace* is implemented as a kernel module for Linux and creates one of the simplest race conditions in parallel programming.

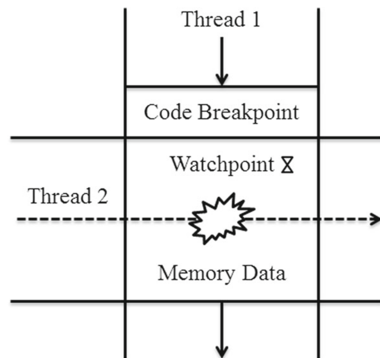

Fig. 1 The basic principle of DRDDR

Fig. 2 The synthetic test case RWrace

Thread 1: write	**Thread 2: read**

`while(ture){`
`...`
S1: `*ptr = rand();`
`...`
`}`
	`while(ture){`
	`...`
	S2: `r = *ptr;`
	`...`
	`}`

As the figure shows, write operation *S1* and read operation *S2* are the key conflicting steps. In this case, we first select a key statement such as *S1* of *Thread 1*, and halt the program at that point by inserting a code breakpoint. Next, we use the decoder to retrieve the memory location of the variable accessed by *S1*, and set a watchpoint at that address, and wait for a certain period of time. If *S2* of *Thread 2* is also active during this time, it will perform a read operation on the shared variable and trigger the watchpoint, and DRDDR will report this event as a data race condition.

DRDDR is ideally suited for data race detection in the kernel for two reasons. First, it is very easy to implement. The basic process, depicted in Fig. 1, is very concise and intuitive. Second, DRDDR has no relation to the synchronization protocols and hardware structures in the kernel environment. This is a distinctive and much appreciated feature of DRDDR in that there is no need to consider the complex semantics of synchronous primitives.

We also note here that not all data races are harmful. In some circumstances, data races have no effect on code execution output. An example is the influence of data race conditions on the update of log/debug variables, which is much more acceptable for programmers. This type of race is termed a benign race [9], and we observe some instances in our experimental results. We discuss these in Sect. 4.

3 Implementation

We describe here our implementation of DRDDR: we use a Linux system based on x86 architecture and we rely on the code breakpoint and watchpoint mechanisms supported in the processor hardware.

The architecture of our solution is depicted in Fig. 3. DRDDR first employs DebugFS [10] as a user interface to initialize each part of the solution. The sampler then selects a small number of memory locations from the large amount of addresses to set code breakpoints. The instructions are analyzed and the decoder is used to access the sampled memory addresses and set watchpoints in those locations. DRDDR searches for potential data race conditions for every sampled memory access instance using a conflict detection algorithm and submits a detailed bug report when operations conclude. It is worth noting that when a data race condition is catched, DRDDR will

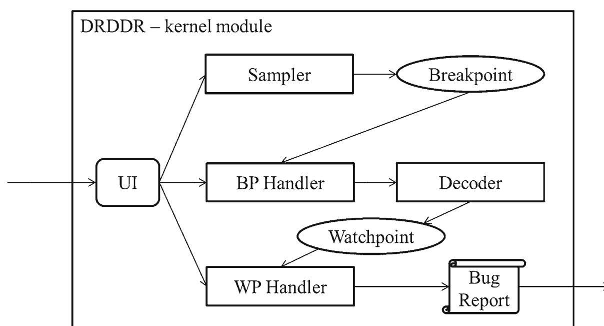

Fig. 3 The architecture diagram of DRDDR

only prints a statement among the program outputs. The formal bug report will be summarized by the detector user according to all the races found in all programs.

Using the watchpoint strategy, two racy threads are identified at the exact start of execution of the conflicting instructions. For this reason, compared to former detection mechanisms, DRDDR is able to collect considerably more useful debugging information. A good example is the stack trace that puts conflicting threads in context, thereby greatly simplifies the analysis of the bug report, and does not add any extra overhead to non-racy programs.

3.1 Sampler

DRDDR performs a basic analysis to statically eliminate those instructions that access only a thread's local stack addresses from the sampling set. Likewise, the solution also deletes instructions that access memory addresses labeled as `volatile` or those that use hardware synchronization primitives. These steps help DRDDR to avoid reporting on races between synchronous variables like locks, the original function of which is to let threads race for the opportunity to access shared data.

The sampler selects a small number of memory access instructions from the sampling set and instruments them with code breakpoints. If any of the breakpoints is triggered, DRDDR runs the conflict detection algorithm on the access operations on the spot, and then randomly selects another program address from the sampling set to insert the breakpoint.

It is worth notice that random sampling of memory access instructions will not lead to inaccurate detection. Because the selectively detection strategy only abandons a portion of memory accessing critical sections, which might miss some races and lead to false negative instances, but as long as DRDDR report a race bug, it is definitely a race condition, with no possibility of false positive case.

As a rule, DRDDR picks program addresses from the sampling set randomly without consideration of local execution frequency. Rarely executed program addresses may therefore be set up with breakpoints after some time, improving the chances of them being scanned for data races.

About the sampling size, we have to mention the detailed structure of debug register in modern CPU architecture: there are over all eight debug registers, DR0–DR7, in which only DR0–DR3 can be used to set watchpoint on memory accesses. Therefore, no matter how the sampling size varies, there are at most four breakpoints being watched at the same time. Besides, as hardware, the relevant operations of debug register all take a very short time, so our data race detection tool has a negligible overhead in parallel programs, and its accuracy will not affected by the sampling size.

To increase the effectiveness of detection results, it is possible to provide DRDDR prior information about which portions of the program may be more likely to cause a data race, perhaps using methods similar to program annotations or pre-analysis. The system may then be configured to give preference to these sections in the sampling process. As the analysis strategy of the instruction execution frequency is not very practicable to apply in our detection tool at present, and we do not know how this

strategy will affect the detecting overhead, we would like to explore its feasibility in the future work.

3.2 Decoder

Given a parallel program encoded in the form of a binary file, DRDDR first uses the decoder to transform it into a sampling set consisting of all instructions that access memory data. This decoding process is accomplished by sending concurrent debug symbols to the binary file. The process may be improved in the future by employing more-complicated decoders.

Regarding the decoder in DRDDR, there are some equivalent choices such as the QEMU [11] emulator and the KVM [12] virtual machine. QEMU is a set of emulators widely used on the GNU/Linux platform that run in both user-mode and system-mode and support many kinds of architectures including IA-32 [13], AMD-64 [14], MIPS R4000 [15], SPARC [16], and PowerPC [17]. However, QEMU is not suited for kernel-mode, where it may encounter many unrecognized libraries and functions, leading to several tricky problems in practice. Furthermore, QEMU has a large volume because of its support for multiple architectures, which causes much inconvenience in transplanting.

In comparison, KVM is a much better match for the requirements of DRDDR. It is an open-source kernel virtual machine with concise functions and good transplant ability. KVM can also run in kernel mode with far less functional dependencies, helping reduce the processing time for certain complex DRDDR functions.

We therefore choose KVM for our implementation and transplant a portion of its features. The decoder in KVM is not a strictly typical one. Since KVM is a virtual machine, its decoder is designed to process only those instructions that cannot be virtually executed in hardware, and therefore supports only a subset of x86 instructions. Our fundamental requirement is analysis of all memory access operations, which clearly does not involve many instructions. We accomplish most of the disassembly functions encountered in experiments but not supported by the KVM decoder.

Our solution also implements a new kernel-user mode interaction module. This facility allows users to obtain DRDDR system information in real time using DebugFS in user mode and the ability to input lists of suspicious addresses using files.

3.3 Interrupt handler

After the sampler selects a small number of program addresses as data race detection candidates, DRDDR sets a watchpoint on the shared variables in the code breakpoint handler. The current thread is then suspended and there is a waiting period to see if any other threads access that particular memory location. Two strategies are used for detection: watchpoint and value-comparison. These strategies complement each other and we describe them in the following sections.

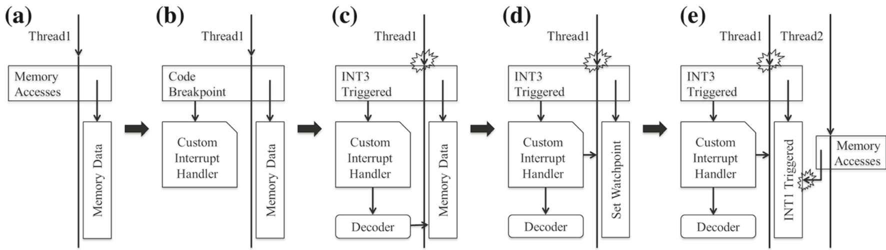

Fig. 4 The basic principle of setting a code breakpoint and watchpoint in DRDDR

3.3.1 Watchpoint

Modern computer architecture has functionality for trapping read and write operations to processor memory addresses, which plays a key role in effectively setting watchpoints in debuggers. DRDDR utilizes the four debug registers provided by x86 hardware to efficiently monitor other memory access operations that may conflict with the current one.

DRDDR uses the code breakpoint and watchpoint functions provided by the debug registers in the following steps, and as detailed in Fig. 4.

1. To detect a memory access instruction, DRDDR will replace the first instruction of its disassemble codes with an INT3 interruption (code breakpoint). The INT3 interruption handler is replaced in turn with a custom manipulation function of our solution for the duration of the subsequent steps (Fig. 4a, b),
2. When the program runs the INT3 interrupt instruction, it will trigger the code breakpoint and invoke the custom handler. This handler uses the decoder we described earlier to decode the current instruction and locate the address of the shared data accessed by the current thread. The INT1 interruption (watchpoint) is added at this point to monitor the data (Fig. 4b–d),
3. If there are any other threads accessing the shared data where the watchpoint has been set, the INT1 interruption will be triggered. DRDDR will consider this memory access instruction to conflict with the original one, constituting a data race, and will report it for further verification and management (Fig. 4d, e).

If the current operation is a write operation, DRDDR sets the processor to the state of trapping both read and write access to the shared data. In the case of a read operation, only the write operations are trapped, because two read access operations to the same memory address do not conflict with each other. If there are no conflict operations detected, DRDDR clears the watchpoint registers and recovers the execution of the current thread.

DRDDR uses an inner-processor interruption to atomically update the breakpoints in all processors because every processor has a single watchpoint register, and this allows for multiple threads that are concurrently sampled in different memory access operations to be synchronized efficiently.

An x86 instruction may access different sizes of memory addresses such as 8-bit, 16-bit, or 32-bit addresses, and DRDDR can set appropriate length breakpoints for

each. If other operations access the critical memory address when the breakpoint is active, they will be trapped by the processor, and this functionality accords with the principles of data race detection. For memory access instructions greater than 32-bit, DRDDR can use at most four registers at the same time to set the breakpoint. If the debug register is not sufficient to detect data races, the value-comparison strategy can compensate.

We consider DRDDR to have succeeded in detecting a data race when the watchpoint is triggered. However, it is more important to catch the race at the site of the conflict when both threads are concurrently accessing the same memory address. The watchpoint strategy in x86 has a disadvantage that when a user enables the paging function, the system performs the breakpoint comparison based on virtual addresses with no other choices. In this case, two memory access operations with the same virtual address but different physical addresses will confound DRDDR.

Kernel threads that access user-level address space in the context of different processes will not conflict with each other. In this case, DRDDR will not use the watchpoint technique but instead choose the value-comparison strategy by default.

Furthermore, if the processors map different virtual addresses to the same physical address, the watchpoint will miss the data race bug, since this technique cannot detect race conditions caused by hardware devices accessing memory directly. The value-comparison strategy is a good solution for these problems.

3.3.2 Value comparison

The value-comparison strategy is very simple: if a write operation conflicts with the sampled operation and modifies the data value at a location in the memory, DRDDR can verify the change by reading the shared data before and after the operation. This strategy has an obvious shortcoming that it cannot detect the situation where a read operation conflicts with a write one. Similarly, it cannot identify conflicting write operations in which the final data value equals the initial value. Even so, the value-comparison strategy is still very effective in practice.

Another issue is that the value-comparison strategy can only catch one of the two racing threads on the spot, which will cause some difficulties in the subsequent debugging process, as we do not know which thread or device accessed the shared data during the monitoring period. This shortcoming is the main motivation and original intention behind using the watchpoint strategy.

4 Experiment and evaluation

There are two main test case resources to evaluate the performance of DRDDR: the synthetic data race micro-benchmark (introduced in Sect. 2) and instances of real data race bugs in the kernel environment. The first resource provides perfect verification of the correctness and functionality of our code at the start of system construction. The latter option better corresponds with the normal execution environment of the kernel system.

After the various processing steps of our solution have gradually been completed, we are in a position to evaluate if DRDDR can perform all its functions fully in a real-world environment. In this section, we discuss real cases along with detailed experimental results.

4.1 Real-world cases

We verify the correctness of DRDDR by screening all data race related bugs for typical, deeply discussed and perfectly solved instances using the official kernel code bugs management tool in Linux, Kernel Bugzilla [18]. Reproducing these bugs involved in locating suitable kernel versions, configuring the relevant environments and writing reproduction scripts. Our experiments show that DRDDR detects these real-world kernel data race bugs accurately and efficiently.

Here we consider in detail two instances that have been discovered in the kernel and already been fixed. One of them occurs in *eCryptfs*, and the other belongs to the *ext2* file system.

4.1.1 eCryptfs

The *eCryptfs* kernel data race is an order violation bug that appeared in kernel version 3.0-rc1 of Linux and was fixed in a subsequent batch update [19].

The bug resides in the *eCryptfs* file system. Figure 5 lists the critical instructions about the race. The virtual file system maintains a single `inode` for each file in memory. When a file is used for the first time, the virtual file system creates a `dentry` for the file that includes its own path and then searches for whether it is in *eCryptfs*. If so, the system creates and initializes a new `inode` and assigns it to `dentry`. The `inode` is locked at the beginning of creation and the *eCryptfs* system needs to unlock it after initialization. However, in this Linux kernel version, the *eCryptfs* system unlocks the `inode` before initializing its size, which leads to other conflict access issues and the opportunity to trigger a data race condition.

To trigger this bug, we require a newly mounted *eCryptfs* file system with one file in it. The system must be brand new to guarantee that the file's `inode` does not previously exist in the kernel. We create a file and use one process to open it, and then conduct a query during this period. Simultaneously another process also opens

Fig. 5 The eCryptfs bug in the Linux kernel

Thread 1:	Thread 2:
ecryptfs_get_inode(){	ecryptfs_write(){
…	
iget5_locked()	…
…// Initialization	iget5_locked()
unlock_new_inode()	i_size_write()
i_size_write()	unlock_inode()
…	…
}	}

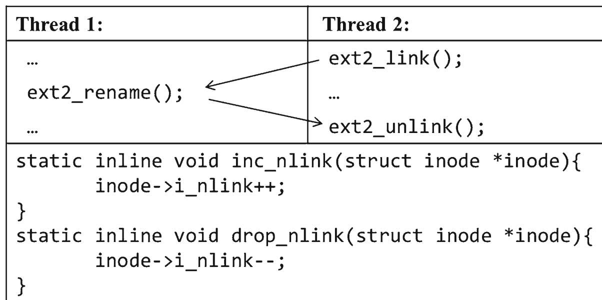

Fig. 6 The ext2 bug in the Linux kernel

the file and writes something to change its size. If the latter action takes place after the unlocking of the query operation and before the initialization of the file size, the data race condition occurs and the size of the file remains unchanged but all contents recorded in the writing process have been lost.

By setting a code breakpoint on the instruction `i_size_write()`, DRDDR can catch the data race successfully.

4.1.2 ext2

The *ext2* kernel data race is an atomicity violation bug. It appeared in kernel version 2.6.38-rc8 of Linux and was fixed in a subsequent batch update [20].

This bug is caused by incorrect usage of the redundant codes that operate the hard link counting of `inode`. Critical steps leading to this bug are depicted in Fig. 6. In the *ext2* file system, the commands `ln` and `link` and system calls `link()` and `linkat()` can be used to create hard links. The content of a file is represented by the `inode` structure and has a member called `i_nlink` to record the number of links that point to itself, which may be queried by `ls -l`. In the function `ext2_rename()` that renames the *ext2* file, the program shifts the simulated links from an old file to a new file and eliminates the links of the old one. The `inc_nlink()` and `drop_nlink()` are inline functions of the link-related functions. Thus, when a file gets a new link, the `i_nlink` of the new `inode` will increase by one while that of old `inode` will decrease by one. However, neither the `ext2_rename()` function nor the function that calls it can lock the old file while updating the link number. This results in the `ext2_rename()` function modifying the `i_nlink` without protection, causing data race conditions with conflicts between the `ext2_link()` and `ext2_unlink()` functions.

DRDDR can detect the race condition by adding a code breakpoint on the instructions that add or subtract `inode->i_nlink`.

4.2 Experimental results

We choose two sets of user mode benchmarks. The first is LTP (Linux Testing Project) [21] from which we extract certain tools such as *fsstress* and *racer*. The

second is UnixBench, which allows us to subject file system codes in the kernel to considerable stress and increase the probability of DRDDR detecting data races. These user mode programs play an important role in reproducing the ext2 bug and identifying new benign data races.

In this project, we construct a set of simple programs, locate all memory access instructions in the user specified functions in the kernel using methods such as regular expression matching and excluding instructions in the stack. The testing processes mainly focus on code in the file system part of the kernel and the analyzing program can locate more than 40,000 memory access addresses (with the actual value depending on the kernel version and compilation options). We apply DRDDR to the Linux kernel to verify its performance in practice and conduct a massive test in which we check for more than 5000 instructions.

In a large number of experiments, DRDDR identified some benign data races. One of these is the kernel function `core_sys_select()` in which there is an instruction that judges if the current thread has some signals still waiting to be processed. During execution, a timer interruption may occur at any instant and the interruption handler will reset the current thread to rescheduling status when it runs out of time slice. These two flag bits belong to the same variable. Therefore, considering only the read-write race aspect, this pair of instructions may be considered a data race. However, what the read-write operation actually involves are different bits of the same variable. Furthermore, because the write operation is accompanied by a bus lock, the two operations cannot occur at the same time though the interruption can enter from any position of the non-interruption codes. We see therefore that this race condition brings no harm to the system and is definitely a benign race.

Another benign race is in the Linux file system pipe read/write processing function `pipe_read()`. We first pick 425 memory access instructions in the file system from all functions that contain “pipe” in their name, and we set 300 code breakpoints in them. We then run a benchmark test program for concurrent pipeline operations from Unixbench for data race detection. One of the memory access operations in `pipe_read()` reads the flag bit of the current task to check if any signals have stalled. When the task stops at this instruction, DRDDR forces it to wait for a short time. During this period a hardware timer interruption comes up and rewrites the flag bit of the stalled signal, creating a race condition. This race is considered benign mainly because it is a read-write conflict and the two instructions access different bits of a single byte.

Prior to these, we also discovered two other benign races using our solution in the Linux kernel. As these are quite similar in principle and only differ in their details, we do not discuss them here.

4.3 Overhead

We give the experimental result of the overhead measurement of DRDDR in this section. As Fig. 7 shows, the X-axis is the average code breakpoints hit number in every second, the Y-axis is the runtime overhead of our detection tool, which is the multiple of the running time with and without DRDDR. From the figure we can see

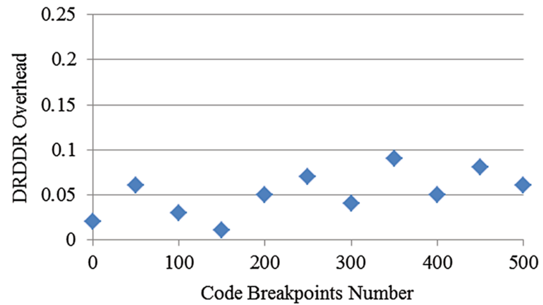

Fig. 7 Time consuming overhead of DRDDR

that the runtime overhead is rarely related to the code breakpoint hit number, which is decided by the sampling size in the detection algorithm. We can also found that the runtime overheads are always less than 0.1 times of the parallel program execution time, which confirm with the conclusion that our detection tool only imports negligible overhead during real time operation.

5 Related work

Existing research most relevant to ours is DataCollider [8], a data race detection solution for the Windows kernel that exploits incident functions of existing hardware in the design of its dynamic technique. The most significant shortcoming of this work is that it is based on the closed-source Windows operating system. This creates some problems for potential users who are eager to use or in dire need of such tools, and researchers who want to conduct further explorations in this area. Considering these restrictions and in the interests of scientific investigation, when we began our efforts, we opted for the most widely used Linux operating system and choose open source tools to implement various functions. We have freely released our solution to the open source community [22], for the benefit of programmers who require such tools, and to assist scholars in related fields to further their research in a much more easier and efficient way.

As DRDDR is a dynamic solution, we introduce other similar work here: there are mainly two ways of detecting data race bugs dynamically, happens-before [23–25] and lock-set [5].

The key principle of the happens-before strategy is to deduce the occurrence sequence of instructions in multi-threaded programs and report potential racy interleaving [26]. The happens-before relation is inferred by monitoring the ordering of inner-thread events and the synchronization primitives between threads. If there are no happens-before relationships between two conflicting instructions, we can conclude that the threads they belong to constitute a racy condition [27]. Though the happens-before strategy introduces no false positives to the concurrency system, the computation in recording and deducing relationships is still very complicated and time-consuming.

The lock-set approach has more advantages if the detected parallel programs are organized using a standard and consistent lock strategy. The main idea is to check

the set of locks owned by each thread when they access shared data and calculating the intersection of the lock sets from certain memory address [28]. If the lock-set intersection is empty, the shared variable is unprotected and prone to produce data races, and the detector reports an issue. This strategy checks for potential violation of shared memory locations, not the real cause of poor instructions in data races, and therefore this approach provides limited assistance in bug detection in parallel programming [29]. Furthermore, the high requirements of the lock-set strategy are not suitable for normal concurrency systems that employ various kinds of synchronization mechanisms in practice.

There are also some hardware-assisted data race detectors such as RACEZ [30], HARD [31], RaceTM [32], and Vulcan [33]. These techniques make use of the Performance Monitor Unit (PMU), Bloom Filter, Hardware Transactional Memory (HTM), and cache coherence protocol transactions in their algorithm designing, respectively. In the last case, the HTM is simulated. The benefit of these solutions is that due to the hardware-based approach, they are all effective and easy to implement. However, none of them is suitable for a kernel environment that has a large number of complex synchronization primitives.

6 Conclusions and future work

In this paper we provide a data race detection technique assisted by an existing hardware structure, the Debug Register in modern x86 architecture. We have implemented our solution on the Linux operating system, the most widely used operating system in the area of systems research. Furthermore, we have experimentally verified its accuracy and efficiency and further elaborated the systematic theory by analyzing, reproducing, and detecting two real-world data race bugs in the Linux kernel environment.

Though the solution emphasizes abetting current practices, it can still prove a useful platform for future exploration. Many interesting directions can be expanded on from here.

The foremost issue is pre-analysis of source code before the detection phase. The breakpoint list in the current implementation is derived completely by extracting all memory access instructions, and the only processing that is ignored are memory access operations using `sp` and `bp`, which operate stack elements. However, most memory access instructions deal with non-shared data, which creates considerable runtime delay. Therefore, enhancing the static analysis of source code could considerably reduce this waste and increase detection efficiency.

Another issue is post-processing of detected data races. After data race bugs have been detected, except for standard registers and stack traces available at present, we should collect more information to assist in debugging [34]. Furthermore, attempting to distinguish harmful data race bugs from benign ones is another possible direction for future work [35]. Similarities among the benign races detected by our solution also raise the questions of how automatic bug detection tools can identify similar bugs.

Finally, considering the overhead problem due to DRDDR, we can improve performance in terms of using more-efficient sampling strategies. Random sampling is simple and easy to implement but has certain shortcomings, including that it is

insufficient in accuracy and creates extra overhead in some cases. Most memory addresses might be irrelevant to data race conditions, and sampling every memory access operation blindly increases the detection overhead. If some frequently performed functions are not involved in data races, but the random sampling strategy coincidentally selects them to insert breakpoints, and this imports a massive amount of unnecessary operations and leads to significant waste of time and resources. If we could obtain information about which portions of a program are more likely to form race conditions, using methods such as program tagging or pre-analysis [36], we could preferentially pick those locations in the sampling process and improve detection performance.

Acknowledgments This work has been partially supported by National High-Tech Research and Development Plan (863 project) 2012AA010901.

References

1. Leveson NG, Turner CS (1993) An investigation of the Therac-25 accidents. Computer 26(7):18–41
2. Poulsen K (2004) Software bug contributed to blackout. Security Focus. http://www.securityfocus.com/news/8016
3. Joab J (2012) Nasdaq's Facebook glitch came from 'race conditions'. http://www.computerworld.com/s/article/9227350
4. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565
5. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T (1997) Eraser: a dynamic data race detector for multithreaded programs. ACM Trans Computer Syst 15(4):391–411
6. Choi JD, Lee K, Loginov A, O'Callahan R, Sarkar V, Sridharan M (2002) Efficient and precise datarace detection for multithreaded object-oriented programs. PLDI'02 37(5):258–269
7. Yu Y, Rodeheffer T, Chen W (2005) Racetrack: efficient detection of data race conditions via adaptive tracking. SOSP'05 39(5):221–234
8. Erickson J, Musuvathi M, Burckhardt S, Olynyk K (2010) Effective data-race detection for the kernel. OSDI'10 10:1–16
9. Baris K, Zamfir C, Candea G (2012) Data races vs. data race bugs: telling the difference with portend. ACM SIGPLAN Notices 47(4):185–198
10. Debugfs-The Linux Kernel Archives. https://www.kernel.org/doc/Documentation/fil-esystems/debugfs.txt
11. Open Source Processor Emulator QEMU. http://wiki.qemu.org/
12. Kernel Based Virtual Machine KVM. http://www.linux-kvm.org/page/
13. IA-32 Architecture. https://software.intel.com/en-us/articles/ia-32-intelr-64-ia-64-architecture-mean/
14. AMD-64 Architecture. http://support.amd.com/TechDocs/24592
15. MIPS R4000 Microprocessor. http://groups.csail.mit.edu/cag/raw/documents/R4400_Uman_book_Ed2
16. SPARC Architecture. http://searchservervirtualization.techtarget.com/definition/SP-ARC
17. PowerPC Architecture. https://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC
18. Linux Kernel Bugzilla. https://bugzilla.kernel.org
19. Ecryptfs bug in Linux kernel. https://bugzilla.kernel.org/id=29752
20. Ext2 bug in Linux kernel. https://bugzilla.kernel.org/id=36002
21. Linux Test Project. http://ltp.sourceforge.net/
22. DRDDR source code on GitHub. https://github.com/ahyangyi/DRDDR
23. Bond MD, Coons KE, McKinley KS (2010) PACER: proportional detection of data races. PLDI'10 45(6):255–268
24. Marino D, Musuvathi M, Narayanasamy S (2009) LiteRace: effective sampling for lightweight datarace detection. PLDI'09 44(6):134–143
25. Flanagan C, Freund SN (2009) FastTrack: efficient and precise dynamic race detection. PLDI'09 44(6):121–133

26. Kaushik V, Chen PM, Flinn J, Narayanasamy S (2011) Detecting and surviving data races using complementary schedules. In: Proceedings of the twenty-third ACM symposium on operating systems principles, pp 369–384
27. Huang J, Meredith PON, Rosu G (2014) Maximal sound predictive race detection with control flow abstraction. ACM SIGPLAN Notices 49(6):337–348
28. Pedro F, Li C, Rodrigues R (2011) Finding complex concurrency bugs in large multi-threaded applications. In: Proceedings of the sixth conference on computer systems, pp 215–228
29. Baris K, Zamfir C, Candea G (2013) RaceMob: crowdsourced data race detection. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles, pp 406–422
30. Sheng TW, Vachharajani N, Eranian S, Hundt R, Chen WG, Zheng WM (2011) RACEZ: a lightweight and non-invasive race detection tool for production applications. ICSE'11 401–410
31. Zhou P, Teodorescu R, Zhou YY (2007) HARD: hardware-assisted lockset-based race detection. In: IEEE 13th international symposium on high performance computer architecture, HPCA 2007, pp 121–132
32. Gupta S, Sultan F, Cadambi S, Ivancic F, Rotteler M (2009) Using hardware transactional memory for data race detection. In: IEEE international symposium on IPDPS 2009, pp 1–11
33. Abdullah M, Qi S, Torrellas J (2012) Vulcan: hardware support for detecting sequential consistency violations dynamically. In: 45th annual IEEE/ACM international symposium on microarchitecture (MICRO), 2012, pp 363–375
34. Jin G, Zhang W, Deng D, (2012) Automated concurrency-bug fixing. In: Presented as part of the 10th USENIX symposium on operating systems design and implementation (OSDI 12), 2012, pp 221–236
35. Narayanasamy S, Wang Z, Tigani J, Edwards A, Calder B (2007) Automatically classifying benign and harmful data races using replay analysis. In: Programming language design and implementation (PLDI '07), pp 22–31
36. Sheng TW (2010) Researches on key technologies of data races detection in concurrent programs. Dissertation for the Doctoral Degree. Tsinghua University, Beijing