An Effective Method to Control Interrupt Handler for Data Race Detection

Makoto Higashi
Graduate School of Information Science and Technology, Osaka University
m-higasi@ist.osaka-u.ac.jp

Tetsuo Yamamoto
College of Information Science and Engineering, Ritsumeikan University
tetsuo@cs.ritsumei.ac.jp

Yasuhiro Hayase, Takashi Ishio, Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University
{y-hayase, ishio, inoue}@ist.osaka-u.ac.jp

ABSTRACT

Embedded software frequently uses interrupts for timer or I/O processing. If a memory area is used by both an interrupt handler and other routines at the same time, the embedded system has the potential to fail because of unexpected data in the memory. To detect the race conditions of memory, this paper proposes a method of interrupt testing on a CPU emulator. The method consists of two features: one is interrupt generation at the instruction points that possibly causes race conditions; the other is replacing input value from external device to control interrupt handlers. An interrupt is generated just after the program reads or writes data on memory for the purpose of covering all possibility of sharing memory between the interrupt handler and other routines. Sequence of input value from the external device is prepared by hand before program execution. We have applied our method to testing for a race condition of *uClinux*. The experience of detecting race conditions has shown the mechanism causes interrupts at necessary and sufficient timing compared with random interrupt testing. Also, it is easy to substitute values in memory to detect race conditions.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-basedsystems**]: Real-time and embedded systems; D.2.5 [**Software Engineering**]: Testing and debugging.Testing tools

General Terms

RELIABILITY

Keywords

race condition, embedded systems, fault injection, interrupt-driven software

1. INTRODUCTION

Embedded software frequently uses interrupts for timer or I/O processing. However, interrupts should be used and tested carefully since wrong use of interrupts causes a race condition [12], i.e. the failure caused by inadequately controlled multiple accesses to shared memory[15]. Assume that a certain variable is changed by an interrupt handler just after a condition check for the variable. In such case, the statements after the condition check should not work correctly since the precondition for the statements may be violated.

To detect the race conditions, static detection methods, which are based on model checking[5, 8, 14], and dynamic detection methods, which intentionally generate an interrupt for testing [16], are proposed. In this paper, we focused on dynamic testing since the dynamic methods generate less false-positive than the static methods.

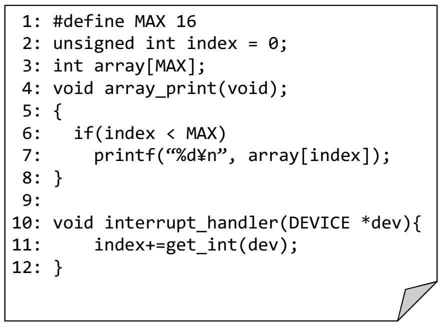
In [16], Regehr proposed a dynamic detection method that randomly causes interrupts while a program is running. This method depends on duration of the random interrupt schedule. If the schedule is too sparse, there are only a little detections of race conditions. If the schedule is too dense, there are always multiple pending interrupts, and then non-interrupt routine cannot make progress. Therefore, it is important to consider adequate interrupt schedule.

At first, we consider the timing in which interrupts occur. To test all race conditions, it needs that an interrupt occurs on all instruction points which access shared memory.

Moreover, it is necessary that appropriate value in memory or variable within interrupt handler to cause a race condition. If there is no execution path to access shared memory, there is no race condition.

In this paper, we propose a method of interrupt testing to detect race conditions. The method comprises two mechanisms. One is that a mechanism to cause an interrupt at all possible timing to cause race conditions. Another is that a mechanism to substitute appropriate value for a value in memory. The former one is to provide a function that an interrupt automatically occurs just after instructions which access shared memory. The latter one is to provide a function that automatically change the value specified by a user. The user specifies memory address and new value in the memory before a program runs.

These two mechanisms are implemented on a CPU emulator to test for race conditions in an early phase of the development of embedded systems. In development of em-



```
 1: #define MAX 16
 2: unsigned int index = 0;
 3: int array[MAX];
 4: void array_print(void);
 5: {
 6:   if(index < MAX)
 7:     printf(“%d¥n”, array[index]);
 8: }
 9:
10: void interrupt_handler(DEVICE *dev){
11:     index+=get_int(dev);
12: }
```

Figure 1: Sample source code that potentially causes race condition

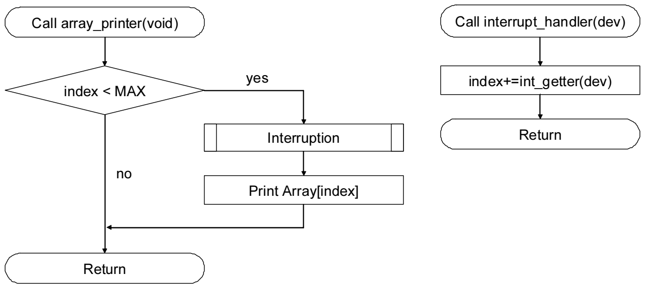


Figure 2: Flowchart of Figure 1 with an interrupt just after the branch

bedded systems, software is often executed and tested on a CPU emulator since hardware and software for the system are developed in parallel. The mechanisms on a CPU emulator enable efficient test for race conditions before the hardware is finished.

The rest of this paper is organized as follows: Section 2 clarifies a race condition that this paper targets. Section 3 describes details of our method, which test for race conditions using the mechanism to cause interrupts and the mechanism to substitute values in memory. Section 4 shows implementation of our method on a CPU emulator. Section 5 evaluates our method through an experiment. Section 6 introduces related work on generating interrupts and changing values. Concluding remark and future work are presented in Section 7.

2. DEFINITION OF RACE CONDITION

In this section, we provide a motivating example, explaining the definition of race conditions in this paper.

Figure 1 shows an example source code that may causes a race condition. Procedure `print_array` checks that `index` is less than `MAX`, then print `index`-th element of `array`.

It apparently seems that `print_array` has no buffer-overrun problem since the function checks the index before it accesses to the array. However, as shown in Figure 2, if an interrupt occurred and `interrupt_handler` is called just after the check of `index`, `index` in line 7 may have `MAX` or more value.

As mentioned above, a program that is made without attention for an interrupt sometimes has an implicit assumption that a variable stores the same value when the variable is accessed previously. Unfortunately, the assumption breaks and the program may not work correctly if an interruption handler asynchronously works and modifies the variable.

In this paper, a race condition is defined as the failure that is caused by interruption handler that modifies a certain variable between a reference or modification to the variable and a later reference to the variable. Our method is designed to detect this kind of race condition.

3. METHOD

This section proposes a method to test for race conditions effectively. The method has two mechanisms. One is the mechanism to cause interrupts and another is to substitute values.

As described in section 2, a race condition occurs if following conditions are met.

1. An interrupt handler is executed when the program is running on the point after an access to a certain variable and before a read access to the variable.
2. The interrupt handler modifies the variable in the execution.

Therefore, we need to treat two tasks in order to detect a race condition; to cause an interrupt between the former access to the variable and the latter reference to the variable, and to control the interrupt handler to access the variable.

Then, the mechanism to cause interrupts treats the former task, and the mechanism to substitute values treats the latter task. Each mechanism runs independently and treats the different task, but both these mechanisms contribute to the solution of the same problem. It is efficient interruption to test for race conditions.

3.1 Mechanism to cause interrupts

This section describes the mechanism to cause interrupts on all point that possibly causes a race condition, i.e. after any access to certain memory area. The mechanism is implemented as a part of memory-accessing instruction in a CPU emulator. The mechanism proceeds in following steps after all memory-access instructions are executed.

1. Checks whether the instruction access memory.
2. If step 1 is true, checks whether the program has just returned from any interrupt handler.
3. If step 2 is false, checks whether the current status of the execution matches the conditions in the configuration file.
4. If step 3 is true, causes an interrupt according to the configuration file.

The step 3 is designed to avoid infinite loop. If interrupts occurs just after returning from interrupt handlers every time, only the interrupt handler runs infinitely. For this reason, the mechanism doesn't cause interrupt in such case.

Figure 3 shows the steps to check whether an interrupt should be caused. Inputs to this mechanism are program counter of a CPU emulator and a configuration file. The configuration file consists of the following items.

kind of interrupt A kind of interrupt which this mechanism causes. An example of this kind is I/O device number linked with the interrupt handler to test for a race condition.

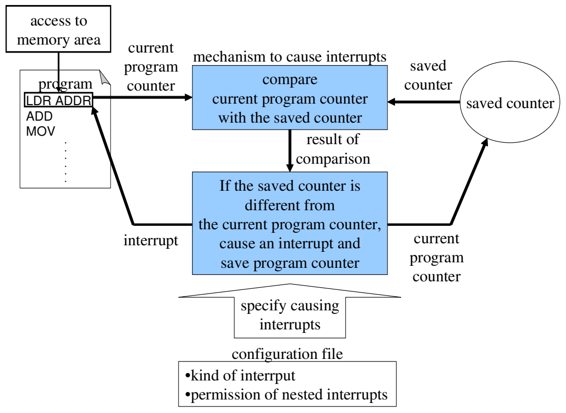

access to memory area
program
LDR ADDR
ADD
MOV
current program counter
mechanism to cause interrupts
compare current program counter with the saved counter
saved counter
saved counter
result of comparison
If the saved counter is different from the current program counter, cause an interrupt and save program counter
interrupt
current program counter
specify causing interrupts
configuration file
•kind of interrput
•permission of nested interrupts

Figure 3: How to cause interrupts.

permission of nested interrupt An option to select whether this mechanism causes an interrupt while the interrupt handler is executed.

The program counter is used for determining whether the program state is just return from an interrupt handler. Program counter is saved just before an interrupt is caused. Then, the saved counter is compared with the current program counter when the mechanism checks to cause interrupt or not. If the two counters are same, the mechanism determines that the program state is just return from the handler.

3.2 Mechanism to substitute values

This mechanism is used for controlling execution path of an interrupt handler by substituting an input value from external device to user specified value. Since embedded software uses memory-mapped I/O in interrupt handlers for communicating with external devices in most case, substituting input value is same as substituting values in a certain memory area.

This mechanism functions on the place where instructions read memory. This is the reason that we assume embedded software on a processor using the memory-mapped I/O. On the processor, a program gets input values from devices by reading memory.

The mechanism proceeds in following steps after all instructions are executed.

1. Checks whether the instruction is reading memory.
2. If step 1 is true, compares the program state to conditions in the configuration file.
3. If step 2 matches, changes the value in the memory according to the configuration file.

Figure 4 shows the steps to substitute a value in a memory area.

Inputs of this mechanism are the memory address the instruction reads and a configuration file. The configuration file consists of the following items.

- memory address: An address which is mapped to external device.

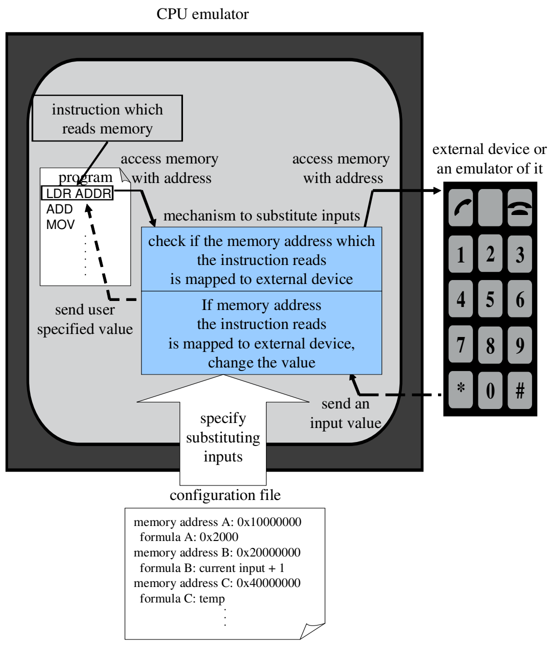

CPU emulator
instruction which reads memory
program
LDR ADDR
ADD
MOV
access memory with address
access memory with address
external device or an emulator of it
mechanism to substitute inputs
check if the memory address which the instruction reads is mapped to external device
If memory address the instruction reads is mapped to external device, change the value
send user specified value
send an input value
specify substituting inputs
configuration file
memory address A: 0x10000000
formula A: 0x2000
memory address B: 0x20000000
formula B: current input + 1
memory address C: 0x40000000
formula C: temp

Figure 4: How to substitute a value.

- formula: An expression to calculate new value from existing value.

You can specify not only constant value, but also an expression using the current value and any other global variables in the expression. This expression makes variety of execution path of interrupt handler possible.

4. IMPLEMENTATION

To automate our method, we implemented a prototype system of our approach on ARM subsystem of SkyEye[3] CPU emulator.

4.1 Mechanism to cause interrupts

We implemented functions to cause interrupts as follows.

- Function to cause interrupts: this is implemented in C language. The function is called by six C functions which deal with load instructions and by six C functions which deal with store instructions.
- Function to detect the return from interrupt handler: when an interrupt occurs, the function saves the value of program counter. To compare current value of program counter with the saved value, we detect the state just after the return from interrupt handler.
- Function to analyze the configuration file: we implemented the function to parse the configuration file by using yacc and lex.In this configuration file, the kind of interrupt consists of I/O device number linked with the interrupt handler to test and whether an interrupt is IRQ or FIQ in ARM architecture.

4.2 Mechanism to Substitute Values

We implemented functions to input new values as follows.

- Function to inject the new value: When load instructions read the value of devices from memory-mapped I/O, we implemented the function which substitute user specified value for the value of devices.
- Function to analyze the configuration file: we implemented the function to parse the configuration file by using yacc and lex.
- Function to get the value of variables: we implemented the function to get information of global variables by using dwarf2[1] before running the program.

5. EXPERIMENT

This section describes a case study in applying our proposed method to test for race conditions for an open-source software. The aim of this study is examine how much reduction of the cost to test for race conditions when a developer doubts the possibility of a race condition. We test for a race condition as a practical matter, and report the process to detect the race condition.

5.1 Target of detection

We select a fault in *uClinux* version 2.4 as the detection target of our method. The fault was fixed in the revision 1.12 of /drivers/char/68328serial.c in the software. [1]

Figure 5 shows an excerpt of the source code which has the fault. Function `rs_flush_chars` is a non-interrupt routine and function `rs_interrupt` is an interrupt handler. These functions are used by the driver of UART port to transmit characters through the port. Both of these functions send one character obtain from the character queue, `info->xmit_buf`, when the following conditions are fulfilled.

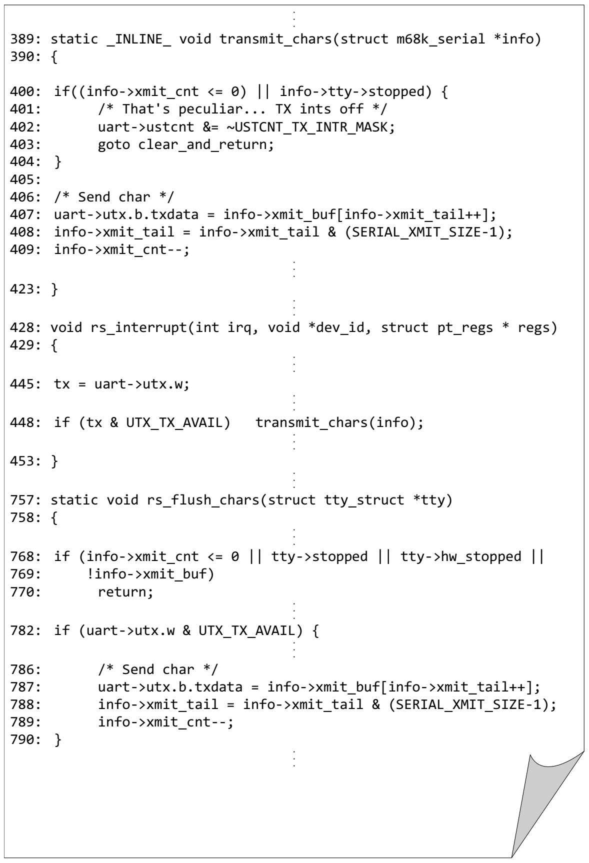
In line 445–448 and 782, UART port is available for transmission, i.e. logical multiply of `utx.w` and `UTX_TX_AVAIL` is true.

In line 400 and 768, The queue stores one or more characters, i.e. `info->xmit_cnt` is greater than 0.

A race condition is exposed if the queue stores only one character and the interrupt handler modifies the queue just after the conditional statement in line 768 branches to else side. Especially, the condition in which the race condition occurs is that the conditional expression in line 768 is false and `info->xmit_cnt` is equal to 1, and `utx.w & UTX_TX_AVAIL` is true. Then last one character is removed from the queue. However, the statements in line 786-789 read a character from the empty queue. Therefore, the program fails and a random character is transmitted.

In this experiment, we modified *uClinux* to run these functions on the emulator.

- *uClinux* is modified to call these functions on ARM architecture, since originally the functions are only called on m68k architecture. The source code of the functions is not modified.



```
                                          ⋮
389: static _INLINE_ void transmit_chars(struct m68k_serial *info)
390: {

400: if((info->xmit_cnt <= 0) || info->tty->stopped) {
401:       /* That's peculiar... TX ints off */
402:       uart->ustcnt &= ~USTCNT_TX_INTR_MASK;
403:       goto clear_and_return;
404: }
405:
406: /* Send char */
407: uart->utx.b.txdata = info->xmit_buf[info->xmit_tail++];
408: info->xmit_tail = info->xmit_tail & (SERIAL_XMIT_SIZE-1);
409: info->xmit_cnt--;
                                          ⋮
423: }
                                          ⋮
428: void rs_interrupt(int irq, void *dev_id, struct pt_regs * regs)
429: {
                                          ⋮
445: tx = uart->utx.w;
                                          ⋮
448: if (tx & UTX_TX_AVAIL)   transmit_chars(info);
                                          ⋮
453: }
                                          ⋮
757: static void rs_flush_chars(struct tty_struct *tty)
758: {
                                          ⋮
768: if (info->xmit_cnt <= 0 || tty->stopped || tty->hw_stopped ||
769:     !info->xmit_buf)
770:       return;
                                          ⋮
782: if (uart->utx.w & UTX_TX_AVAIL) {
                                          ⋮
786:       /* Send char */
787:       uart->utx.b.txdata = info->xmit_buf[info->xmit_tail++];
788:       info->xmit_tail = info->xmit_tail & (SERIAL_XMIT_SIZE-1);
789:       info->xmit_cnt--;
790: }
                                          ⋮
```

Figure 5: Excerpt of the failure in uClinux-2.4.x/drivers/char/68328serial.c rev 1.11

- *uClinux* is modified not to call function `__delay`, since this function wastes much execution time. It is not related with both function `rs_flush_chars` and function `rs_interrupt`, and the evaluation of this experiment does not need an exact execution time.

Also, in this experiment, we did not select the option explained in section 3.1 that chooses to cause interrupts while the interrupt handler to test is being executed. This is because a race condition can be caused without multiple interrupts, and a large amount of multiple interrupts prevent a non-interrupt routine from making progress.

5.2 Testing process

We report the process to detect the race condition we described in the above. It must be noted that a developer who tests for the race condition has following knowledge. He or She:

- Doubts the possibility to cause a race condition within function `rs_flush_chars`.
- Assumes function `rs_interrupt` as the interrupt handler to cause the race condition in cooperation with function `rs_flush_chars`
- Assumes variable `info->xmit_cnt` as a variable to cause the race condition.

[1] `http://cvs.uclinux.org/cgi-bin/cvsweb.cgi/uClinux-2.4.x/drivers/char/68328serial.c.diff?r2=1.12&r1=1.11&f=h`

He or she must proceed following procedures to test effectively.

- Examines adequate number of data in `info->xmit_buf` `queue` before calling function `rs_flush_chars`. The number depends on the number of interrupts are caused by our method just before calling function `rs_flush_chars` and from 757 through 767 in Figure 5. Because the number of `info->xmit_buf` must be at least one in the state the program executes line 768 in Figure 5 to cause the race condition. Also, the developer considers that an interrupt occurs just before calling function `rs_flush_chars` and from line 757 through 767. Therefore, the number of `info->xmit_buf` must be calculated to use above information.
- Describes the number in the configuration file.

Also, he or she must find out following information that is required in the configuration file.

- Memory address of `utx.w` member in structure `uart` that represents UART port: This address is required to decide the value of `uart->utx.w`. There are two ways to investigate the address. One is to check the specification of UART port. Another is to find out the operand of load instruction which reads the value of `uart->utx.w` to use a debugger. In the latter way, the method to find out the load instruction is to search the address of all lines in source code by using dwarf2[1].
- A value of `uart->utx.w` to cause the race condition: The value is decided to investigate the value of `UTX_TX_AVAIL` of line 445–448 and 782 in Figure 5.
- A kind of interrupt and interrupt number to call function `rs_interrupt`: These are decided to find out the line of function `rs_interrupt` call instruction in the interrupt vector of source code.

5.3 Result

We show the cost of testing for race conditions by the process we explained in section 5.2

It is needed to test more than six data about the value of `info->xmit_buf` to detect the race condition. The reason is to cause the race condition before calling function `rs_flush_chars` only when the number of `info->xmit_buf` is from 6 through 10.

If the number is less than 6, our mechanism causes interrupts 5 times before executing line 768 in Figure 5. In such case, `info->xmit_buf` is empty at the execution of line 768. As a result, lines 787–789 are not executed. On the contrary, if the number is more than 10, the number of `info->xmit_buf` is more than 1 at the execution of line 787–789. As a result, there is no race condition.

A developer required writing 7 lines in the configuration file to test for a race condition in this case. The amount of the file is small. However, to get information to describe the file, it took about 10 minutes to search variables in source files of *uClinux-2.4.x* and dynamically find out addresses of variables in *uClinux-2.4.x* executing on the CPU emulator.

Next, we measured CPU cycles from the start of the CPU emulator through detecting the race condition. It took 7241-7488 cycles to execute *uClinux-2.4.x* on the CPU emulator in applying our method. On the other hand, it took 4836078 cycles to normally execute *uClinux-2.4.x* on the CPU emulator. It requires time about 15 times to apply our method. Other information is as follows.

- The number of interrupts: 1409375
- The number of calling the interrupt handler: 390722
- The total number of cycles took in the interrupt handler:69952632 (96% of all cycles)

5.4 Discussion

We argue the following points about the experience and the results of it.

- difference from random interrupt testing
- how to prepare values of variables
- knowledge about target program

Difference from random interrupt testing.

The aim of our method is to improve random interrupt testing(Regehr's method) [16] as we explained in section 1. There are two points to improve it, a timing of an interrupt and behavior of an interrupt handler.

First, we argue the timing of an interrupt. In our method, an interrupt occurs just after an instruction accesses memory. On the contrary, interrupts randomly occurs in Regehr's method. The timing of an interrupt between these methods does not correspond. However, a user can set up the schedule of interrupts in Regehr's method. If the schedule is too dense, the timing of interrupts by Regehr's method includes the timing of interrupts by our method. However, we think it is difficult that our method and Regehr's method correspond.

The reason is that Regehr's method cannot detect nested interrupt handler call. In our method, a user can select a permission of nested interrupts as we described in section 3.1. This option prevent over interrupt. On the other hand, Regehr's method cannot prevent multiple interrupts.

Next, we argue the behavior of an interrupt handler. Regehr's method can only trigger interrupts. Our method can not only trigger interrupts but also substitute the value of variables within the interrupt handler. Using this mechanism, a user can control execution path of the interrupt handler. As a result, it is possible to try to test more case to cause race conditions.

Given this mechanism, our method Regehr's method are different kinds of testing. In our method a user have to understand the behavior of the program. On the other hand, a user does not have to understand it in Regehr's method. We consider the difference between our method and Regehr's method as the relation between white-box testing and black-box testing.

It is useful to be able to change execution path of the interrupt handler to detect race conditions. An interrupt from external device occurs and reading and/or writing values from the device are a close relationship. Therefore, to change the value from the device has an impact to the behavior of the interrupt handler.

However, a user needs to know information of the device and values of it in advance. If there is no information, a user cannot specify the value. The difference between our method and Regehr's method is small in such case. A future work is to test by using as little information that a user needs to describe as possible.

On the other hand, users of our approach have to determine which combinations of an interrupt handler and a non-interrupt routine should be tested. The following steps for all interrupt handlers enumerate all the combinations that potentially cause race condition.

1. Detect all variables to which an interrupt handler writes.
2. For each one of the variable, find all routines that access the variable at least twice.

How to prepare values of variables.

Information of not only interrupts and input values from devices, but also values of variables in the program may be required to detect race conditions as we described in section 5.3. We discuss how to prepare the information.

In section 5.3, we substituted values of variables by modifying the source code of *uClinux* to inject the information. Then, we wasted much compile time because *uClinux-2.4.x* is a large scale of software, As the result, the testing for race conditions was inefficient.

It is realized to use the mechanism to substitute values in memory to easily inject the information. The aim of the mechanism is to substitute appropriate value for a value from extern device. You specify memory-mapped I/O address of the device to use the mechanism. If you specify the address of the variable, you can change the value of a variable in source code.

It is generally difficult to investigate the address of variables. However, it is easy to know the name of the variable. We have to extend the format of a configuration file to be able to write the name instead of the address.

Knowledge about target program.

In section 5.2, we assumed a user knows non-interrupt routine, interrupt handler and shared memory of target program that test for race conditions in advance. We argue the knowledge of a user to test for race conditions.

First, we assume that a user has little knowledge about target program, but enough about external device. A user can test only when race conditions occur whenever an interrupt occurs. Because only knowledge of a kind of device is sufficient information to cause interrupts exhaustively.

Next, we assume that the user has knowledge about values from an external device. The user may be able to select the value to execute an instruction that causes race conditions. Executing the path needs to examine source code in a precise sense. However, it is possible to detect race conditions only to create values according to the specification of the external device.

To detect all race conditions is needed to create adequate test cases. It is important that combination non-interrupt routine with interrupt handler to cause a race condition. Therefore, to cause all race conditions requires checking all execution paths of not only interrupt handlers but also non-interrupt routines. Also, there is a particular kind of case as section 5.3. To detect a race condition is to make only 6 sets of test cases in such case.

However, it is impossible to detect race conditions in the

```
 1: unsigned int len = 0;
 2: void str_cpy(char *buf, char *str);
 3: {
 4:   len = strlen(str);
 5:   if((0 < len) && (len <= strlen(str)))
 6:     memcpy(buf,str,len+1);
 7: }
 8:
 9: void interrupt_handler(void){
10:     len++;
11: }
```

Figure 6: Sample source code that causes race condition when interrupt occurs only once

following conditions. As a future work, we are trying to detect these race conditions.

- A race condition that depends on the number of interrupts or a combination of interrupts: For example, when an interrupt is not caused just after first load instruction and is caused just after next load instruction, a race condition occurs. Our method cannot detect the race condition. For example, in Figure 6, race condition occurs when an interrupt does not occur in line 4 and occurs in line 5.
- A race condition that depends on a combination of multiple kinds of interrupts: For example, when at first a timer interrupt is caused, next, UART interrupt is caused, a race condition occurs. Our method deals with only one kind of interrupt at a time.

If a user needs to test multiple interrupt handlers, the user needs to apply our method with respect to each interrupt. In such case, the number of execution of the program to test is multiplying the number of interrupt handlers by the number of test case. Test case means a value from external device to cause a race condition.

6. RELATED WORK

Testing embedded software with hardware interrupt such as device drivers is challenging because developers must investigate possible race conditions on a large number of control-flow paths caused by interrupt.

Regehr proposed a method to test interrupt-driven embedded software by causing interrupts at random [16]. Our approach uses a CPU emulator to exhaustively cause interrupts to cover possible control-flow paths.

Although our research focuses on interrupt-driven embedded software, detecting a race condition is also an important issue for developing multi-threaded programs. Entropy Injection[4] inserts artificial delays between data access operations to provoke race conditions in multi-threaded programs. This approach increases the probability of race conditions during an automated test session. Our approach controls the interrupt mechanism of a CPU emulator to directly cause race conditions; our approach allows developers to test and debug a particular execution scenario of a program under test.

Joshi et al. [11] proposes a method extracting a cyclic lock dependence chain from an execution trace of a multi-threaded program; their customized thread scheduler suspends threads that acquire locks in the cycle to create a deadlock with high probability. Our tool controls schedule of interrupts instead of threads to create data race conditions. A difference between interrupt-driven embedded software and multi-threaded program is that an interrupt handler often reads and writes shared variables and most of them do not cause any problem. Data race conditions of interest to us are caused by statements executed under assumptions on values stored in variables that might be updated by an interrupt handler.

There are also several dynamic analysis methods [9, 13]; however, these methods cannot be applied to interrupt-driven software. Regehr and Cooprider have proposed an interesting approach that translates source code of an interrupt-driven program into a multi-threaded program that emulates the original behavior [17]. We did not take this approach because we would like to test an interrupt-driven program on a CPU emulator to analyze the actual behavior of the program when a race condition occurs.

Our approach monitors a program execution, artificially causes interrupts and generates input data from external devices when the program accesses a particular memory address. To implement our method, we have employed a fault injection mechanism. Although the main goal of fault injection research is to decide input values that likely cause failures of systems to test and evaluate a program [10], a fault injection approach that dynamically substitutes a modified data for an actual data with a modified data is well suited for us to generate input data to analyze a race condition.

Xception[6] is a tool to emulate hardware transient faults in functional units of a processor. For example, Xception can corrupt an operand loaded from a specified address with several bit level operations: stuck-at-zero, stuck-at-one, bit flip, and bridging. Our method implements a similar mechanism but substitutes an arbitrary value for an input value specified by a developer so that developers can test a particular execution scenario.

Qinject[7] is a CPU emulator with another fault injection approach. Qinject generates input values when a specified instruction is executed in order to emulate a defective code, while our method generates input values when instructions access a specified memory location in order to emulate a memory mapped I/O.

To test and debug a program, `gdb`, the GNU Project Debugger [2], also can substitute a test input for an actual input when an instruction loaded the input from a specified memory address. However, `gdb` cannot directly control interrupt. We have implemented our tool on a CPU emulator instead of a combination of a regular CPU emulator and `gdb` because of the ease of implementation.

7. CONCLUSIONS

In this paper, we have implemented two mechanisms to a CPU emulator, which can cause interrupts after memory accesses and substitute input values from external devices that interrupt handler uses to test for race conditions in embedded software. We have applied our method to testing for a race condition of *uClinux*. The experience of detecting race conditions has showed the mechanism causes interrupts at necessary and sufficient timing compared with random interrupt testing. Also, it is easy to substitute values in memory to detect race conditions.

Future works will focus on substituting value of variables in source code, causing more appropriate timing of interrupts and dealing with multiple kinds of interrupts.

8. ACKNOWLEDGMENTS

This research was supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (A) (No.21240002). This research was also supported by Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B) (No.21700031).

9. REFERENCES

[1] Dwarf home. `http://dwarfstd.org/`.

[2] Gdb: The GNU project debugger. `http://www.gnu.org/software/gdb/`.

[3] SkyEye - open source simulator. `http://www.skyeye.org/`.

[4] L. Albertsson. Entropy injection. *SICS Technical Report*, T2007(2), February 2007.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 44(4):73–85, October 2006.

[6] J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998.

[7] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Qinject: A virtual machine based fault injection framework. International Conference on Architectural Support for Programming Languages and Operating Systems (Poster Presentation), March 2008.

[8] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review*, 37(5):237–252, December 2003.

[9] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, June 2009.

[10] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, April 1997.

[11] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 110–120, June 2009.

[12] N. G. Leveson. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[13] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–143, June 2009.

[14] E. G. Mercer and M. D. Jones. Model checking machine code with the GNU debugger. In *12th International SPIN Workshop*, pages 251–265, August 2005.

[15] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[16] J. Regehr. Random testing of interrupt-driven software. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 290–298, September 2005.

[17] J. Regehr and N. Cooprider. Interrupt verification via thread verification. *Electronic Notes in Theoretical Computer Science*, 174(9):139–150, June 2007.